Universidade do Minho
Escola de Ciências

Nuno David Da Costa Martins

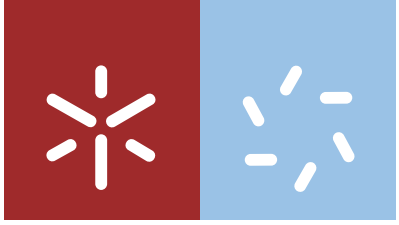
Programação em R no estudo de probabilidades

outubro de 2016

Programação em R no estudo de probabilidades

Nuno David Da Costa Martins

UMinho | 2016



Universidade do Minho
Escola de Ciências

Nuno David Da Costa Martins

Programação em R no estudo de probabilidades

Dissertação de Mestrado
Mestrado em Estatística

Trabalho realizado sob orientação da
Doutora Cecília Castro Azevedo

outubro de 2016

Declaração

Nome: Nuno David Da Costa Martins

Endereço eletrónico: nunomartins_11@hotmail.com

Telemóvel: 936981195

Cartão de cidadão: 13615566

Título da dissertação de mestrado:

Programação em `R` no estudo de probabilidades

Orientadora:

Doutora Cecília Castro Azevedo

Ano de conclusão: 2016

Mestrado em Estatística

Universidade do Minho, 31 de outubro de 2016

Agradecimentos

À Doutora Cecília Azevedo, pela compreensão, paciência, amizade e sobretudo conhecimento transmitido desde o início deste trabalho.
Ao departamento de vendas e informática dos *Hoteis Fenix*, pelo bom ambiente de trabalho, pela compreensão e flexibilidade em complementar os dois trabalhos.
A todos os meus amigos, pela companhia, apoio e bons conselhos. Em particular a Jorge Vieira, Fernando Gomes, Jorge Camarinha, Carla Silva, Duarte Maia, Manuel dos Prazeres, Rui Santos, Marta Rodrigues, entre outros.
Em especial, à minha namorada, Marisa Linhares, pelo carinho, bondade, incentivo e sobretudo paciência, sempre presentes.
Por último, um agradecimento profundo à minha família, pela educação e valores transmitidos, em particular, à minha mãe, irmãs e ao meu PAI, de quem tenho uma enorme admiração e saudade.

Obrigado a todos.

Resumo

A linguagem de programação do *software* estatístico `R`, inicialmente desenhado para a análise de dados estatísticos, é hoje utilizada por inúmeros investigadores e cientistas em todo o mundo.
O objetivo deste trabalho consiste em explorar esta linguagem de programação e, usando diversas potencialidades herdadas de vários paradigmas, definir o espaço de probabilidades associado a uma versão do jogo de *Poker*. Para a definição deste espaço são utilizadas diversas funções existentes no *package* `prob`, tendo sempre em conta a simplicidade dos métodos. Adicionalmente, são apresentadas simulações do jogo, onde a partir da extração de cartas de um baralho comum, com base nas regras do jogo e na definição das *mãos*, se decide um vencedor.

Abstract

The programming language of statistical *software* `R`, inicially designed for data analysis, is today used by numerous researchers and scientists in all world.
The main goal of this work is to explore this programming language and,using some inherited potential of several paradigms, defining the probability space associated with a version of the *Poker* game. For the definition of this space, are used several functions from the `prob` *package*, always having in mind the simplicity of processes. Aditionally, are presented simulations of the game, where from the extraction of cards from a single deck, based on the rules of the game and the definition of the *hands*, decide a winner.

Índice

Índice de tabelas

Índice de figuras

Capítulo 1

Introdução

O R é um programa que teve, na sua génese, como grande objetivo o tratamento de coleções de dados, razoavelmente extensas, permitindo análises estatísticas ricas, uma vez que inclui um sem número de programas, em *packages*, que se aplicam a objetos de diferentes classes, assim como a dados em diversos formatos e tipo. Além disso, o R pode ser usado como *interface* com outro *software*.

O R é, também, uma poderosa linguagem de programação, o que tem facilitado, e permitido, a sua extensão a diversas áreas da matemática. A teoria das probabilidades é uma delas.

Uma linguagem de programação, por ser munida de gramática, ou seja, sintaxe específica (que permite a definição de instruções), exige raciocínio lógico-matemático fundamental na compreensão (e facilitador na aplicação) de conceitos e resultados que podem ser bastante complexos.

A linguagem R tem incorporada vários paradigmas de programação, tais como imperativa, funcional e orientada a objetos. Esta linguagem resulta da combinação destes paradigmas, utilizando diversas bibliotecas de programação já existentes noutras linguagens. Estes paradigmas, tornam-se especialmente relevantes quando as aplicações são complexas ou a qualidade do *software* resultante é importante. Em particular, as versões de programação orientada a objetos em R podem ajudar a lidar com a complexidade de certos tipos de dados subjacentes (Chambers (2008)).

A necessidade de lidar com *terabytes* de informação requer o recurso a ferramentas de programação cada vez mais versáteis e poderosoas. Assim, o *software* R tem sofrido

algumas transformações, de modo a acelerar computações em grande escala e, complementarmente, criar um *interface* com outras linguagens quando estas fornecem melhor desempenho em aplicações computacionalmente exigentes, como, por exemplo, em *Big Data*. Nesta questão, o paradigma funcional é fundamental pois permite que funções criadas de forma independente, por não terem qualquer ambiguidade, sejam utilizadas em diversos programas.
Tanto a programação orientada a objetos como a funcional enquadram-se, naturalmente, em aplicações estatísticas e em `R`. De forma breve, pode dizer-se que a programação funcional é útil em implementar funções confiáveis e executáveis para diferentes modelos e em diferentes plataformas. Já a programação orientada a objetos disponibiliza ferramentas para uma melhor definição dos objetos do modelo e para uma adaptação a novas ideias e novos modelos (Chambers (2008)).
Os princípios fundamentais da programação funcional podem ser caracterizados da seguinte forma:

- Programar consiste em definir funções;
- A definição de função numa linguagem de programação funcional, é a mesma de função em matemática, isto é, retorna um único valor correspondente a cada um dos conjuntos de argumentos válidos, dependente apenas desses argumentos e não havendo efeitos colaterais que possam alterar outros cálculos.

As ideias principais da programação orientada a objetos podem ser descritas da seguinte forma:

- Tudo o que é computado é um objeto e objetos devem ser estruturados de acordo com o objetivo dessas mesmas computações;
- A ferramenta de programação chave é a definição de uma classe, onde objetos pertencentes a esta classe partilham uma estrutura definida por todas as propriedades que apresentam, sendo essas propriedades objetos de uma classe específica;
- De uma classe específica pode derivar uma subclasse, de tal modo que um objeto dessa subclasse, seja também um objeto da classe inicial.

Pretende-se, neste trabalho, apresentar com detalhe a programação de uma versão do jogo de *Poker*, recorrendo a diversas funções (funcionais, predicados) já implementados

em *packages* do `R`, usando como base de trabalho o *package* `prob`. Esta programação encontra-se no Capítulo 4 do presente trabalho. O resto do documento encontra-se organizado da seguinte forma:

- Capítulo 2 - Objetos e Funções especiais, onde são explorados objetos de diversos tipos, e, em particular, funções já definidas no programa. São ainda apresentadas funções especiais para o tratamento e manipulação de *strings*.
- Capítulo 3 - O *package* `prob`, onde são apresentadas e exploradas as principais ideias constantes nesta livraria.
- Capítulo 4 - O jogo de *Poker*
- Capítulo 5- Conclusão

Capítulo 2

Objetos e Funções especiais

Um programa é um conjunto de instruções. Numa linguagem como o R, estas instruções são funções ou expressões.
Uma função pode ser definida como parte de um código escrito para realizar uma tarefa específica, podendo admitir determinados argumentos ou parâmetros. Os argumentos são o *input*, os resultados obtidos o *output*.
Neste tipo de linguagem, quer a função, *input* ou *output* são objetos de determinada classe, o que permite que se efetuem operações sobre as mesmas de uma forma particular dependendo da classe e tipo de objetos em causa.
Este Capítulo contém uma breve descrição dos diversos tipos de objetos e funções mais utilizadas neste trabalho. São apresentados exemplos sempre que exista necessidade de esclarecer ou clarificar o conceito em causa. São ainda apresentadas funções especiais para a manipulação de *strings*.

2.1 Objetos em R

A linguagem de programação R suporta o paradigma "orientada a objetos", pelo que as variáveis, dados, funções, resultados, etc., são guardados na memória ativa do computador como um objeto, com um nome específico.
Estes objetos podem ser manipulados com operadores (aritméticos, lógicos ou comparativos) ou funções (que também são objetos).
As estruturas de dados mais frequentes nos objetos são as variáveis, vetores, matrizes,

fatores, variáveis indexadas, cadeias de caracteres, listas e *data frames*.

As estruturas de dados mais simples, variáveis ou dados, podem ser da classe `numeric`, número real em dupla precisão que podem ser escritos como inteiros, com fração decimal ou em notação científica, `complex`, números complexos, `logical`, variáveis lógicas `TRUE` ou `FALSE`, ou `character`, cadeias alfanuméricas (*strings*). Existem ainda outros dados especiais, tais como `NA`, `Inf` ou `NAN`, que designa valores omissos (*Not Available*), infinitos e *Not a Number*.

Os objetos podem ser atómicos ou recursivos. Nos objetos atómicos, todos os elementos que os compõem são do mesmo tipo (ou `mode`). Exemplos destes objetos são os da classe `vector`, `array` ou `matrix`. Os objetos recursivos combinam uma coleção de outros objetos de diferente tipo. É o caso de objetos da classe `data.frame` ou `list`.

Existem, no entanto, outras formas recursivas importantes. É o caso dos objetos da classe `function`.

Todos os objetos têm uma determinada estrutura e atributos. Os atributos de um objeto fornecem informação específica sobre o próprio objeto. O tipo (ou `mode`) é um exemplo de um atributo do objeto. Todos os objetos têm dois atributos intrínsecos: tipo e comprimento (`mode` e `length`). Mesmo um objeto vazio tem `mode` e `length`.

Exemplo 2.1.1.

```
v<-numeric()
mode(v)
[1] "numeric"

length(v)
[1] 0

w<-NULL
mode(w)
[1] "NULL"

length(w)
[1] 0
```

No entanto, existem atributos que nem todos os objetos possuem. Por exemplo, o atributo `levels` é exclusivo dos objetos da classe `factor`. A estrutura de um objeto pode ser consultada com a função **`str()`** e os atributos com a função **`attributes()`**, mas nem sempre se tem essa informação, apesar de todos os objetos terem, como foi dito anteriormente, dois atributos intrínsecos.
Em programação orientada a objetos, são os atributos que definem o contexto para a execução de um comando e, consequentemente, o seu resultado, ajudando a descrever um objeto. Por exemplo, os nomes das colunas num objeto da classe `data.frame` ajuda a perceber que tipo de dados estão contidos em cada uma das colunas.

De seguida, encontram-se descritos alguns objetos do R, bem como os respetivos atributos:

- `vector` - Objeto atómico.

Todos os seus elementos são do mesmo tipo. Um vetor composto por valores numéricos pode pertencer à classe dos números reais (`numeric`) ou dos inteiros (`integer`). Já um vetor composto por caracteres ou sequências de caracteres alfanuméricos pertence à classe `character` ou `factor`. A função mais básica para definir vetores é **`c()`**.
É possível "forçar" um objeto a pertencer a outro tipo por *coerção explícita*. Para tal existe a família de funções **`as.*()`**.

Exemplo 2.1.2.

```
x<-0:7
class(x)
[1] "integer"
as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6" "7"
```

Por vezes, quando são aplicadas coerções não permitidas, é imprimida uma mensagem de aviso, transformando os elementos em valores omissos (`NAs`), como se pode observar no exemplo seguinte:

Exemplo 2.1.3.

```
x<-c("a","b","c")
asnum<-as.numeric(x)
```

output:

```
Warning: NAs introduced by coercion
[1] NA NA NA
```

Da análise do *output* do Exemplo 2.1.3, verifica-se que não é possível converter o objeto `x`, da classe `character`, num objeto do tipo numérico.

- `arrays`: variáveis indexadas. Objeto atómico.

Uma variável indexada, `array`, é um objeto atómico com um atributo adicional, `dim`, o qual, por sua vez, é um vetor numérico de dimensões, números inteiros positivos, formado por vários índices. Vetores e matrizes são casos particulares.
Os elementos do vetor de dimensões indicam os limites superiores dos índices. Os limites inferiores são sempre 1.
Para criar uma variavel indexada é suficiente atribuir a um vetor o atributo `dim`.

Exemplo 2.1.4.

```
z<-numeric(1500)
class(z)
[1] "numeric"
dim(z) <- c(3,5,100)
class(z)
[1] "array"
```

Pode ainda utilizar-se a função **`array()`**, tendo como argumentos um vetor de dados e um vetor de dimensões.

Exemplo 2.1.5.

```
h<-numeric(24); Z<-array(h, dim=c(3,4,2))
```

output:

```
Z
, , 1

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0

, , 2

     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    0    0
[3,]    0    0    0    0
```

Caso se pretenda referir a um elemento de um `array`, recorre-se ao seu nome e, entre parêntesis retos, os respetivos índices separados por vírgulas. No *output* do exemplo anterior, verifica-se que foram devolvidas duas matrizes 3×4. De facto, deixando livres o primeiro e segundo índices, o terceiro fixado em 1 ou 2, devolve uma matriz (`array` com vetor de dimensão de comprimento 2).
Se se fixar o segundo índice, obtém-se uma matriz 3 por 2 (no total existem 4). Fixando o primeiro índice é obtida uma matriz 4 por 2 (no total existem 3). Fixando os 3 índices, obtém-se um valor numérico (no total existem 24).

Exemplo 2.1.6.

```
Z[ ,3, ]
     [,1] [,2]
[1,]    0    0
```

```
[2,]    0    0
[3,]    0    0

Z[2 , , ]
     [,1] [,2]
[1,]    0    0
[2,]    0    0
[3,]    0    0
[4,]    0    0
Z[2,3,2]
[1] 0
```

Além disso, a definição da variável indexada tem como regra que o primeiro índice é o que se movimenta mais rapidamente e o último é o mais lento.

Exemplo 2.1.7.

A variável indexada atrás definida, `Z`, tem $3 \times 4 \times 2 = 24$ elementos que se formam a partir dos elementos originais na ordem:

`Z[1,1,1], Z[2,1,1],...,Z[2,4,2], Z[3,4,2]`

As matrizes são um caso particular de um `array` com duas dimensões em que o atributo adicional relativo à dimensão é um vetor numérico de comprimento 2, correspondente ao número de linhas e ao número de colunas da matriz. Todos os seus elementos são do mesmo tipo.
É possível obter uma matriz com o comando **`matrix()`**. Por defeito a matriz é definida por colunas. É possível atribuir nomes às suas colunas usando o atributo `dimnames`. As funções **`is.matrix()`** e **`as.matrix()`** comprovam, ou forçam, o caráter de matriz a um objeto.

- `list`: objeto recursivo

Consiste numa coleção ordenada de objetos, os componentes. Estes componentes não têm que ser do mesmo tipo, nem comprimento. Uma lista pode ser composta, a título de exemplo, por um vetor numérico de comprimento 5, um valor lógico, um vetor de

caracteres de comprimento 2, uma matriz de dimensão 7 por 3, etc.
Os componentes de uma lista estão sempre numerados. Podem ser chamados pelo seu número ou nome (que não é obrigatório). A seleção dos componentes de um objeto da classe `list` é feita a partir de `[[ ]]`, ou usando **$** caso os componentes tenham nome. Um cuidado a ter é que os parêntesis retos simples, `[ ]`, retornam uma sublista e não os componentes da lista. Este objetos podem ser criados com a função **`list()`**.
A função **`length()`** aplicada a uma lista, devolve o número de componentes da mesma.

- `data.frame`: objeto recursivo

Este tipo de objeto é um caso particular de uma lista, cujos componentes têm o mesmo comprimento. São apropriados para descrever "matrizes de dados", onde cada linha representa um indivíduo e cada coluna uma variável. Estas variáveis podem ser quantitativas ou qualitativas. Se forem qualitativas estão sempre na classe `factor`.
Ao contrário de um objeto da classe `matrix`, um `data.frame` pode armazenar objetos de diferentes classes em cada coluna. Além dos nomes das colunas (obrigatórios), indicando os nomes das variáveis, este tipo de objeto, à semelhança do objeto `matrix`, tem o atributo especial `row.names` que atribui nomes à linhas.
Por serem tabelas de dados construidas com vista a tratamento estatístico, estes objetos podem ser carregados a partir de ficheiros existentes em ambientes exteriores ao `R`. Cada vez é mais fácil importar (e exportar) ficheiros de outros programas. Em particular, o *package* `foreign` contém funções específicas para importação a partir de outros programas estatísticos tais como *SPSS, SAS, STATA, Minitab, SPlus, EpiInfo, Systat* e também programas numéricos, como, por exemplo, *Octave.*
Claro que é possível definir um `data.frame` diretamente no `R` com, por exemplo, a função **`data.frame()`**. É ainda possível usar *coerção explícita* a partir de outros tipos de objetos, como listas (**`as.data.frame()`**). De facto, estes objetos são listas em que os componentes são as colunas (assim como as matrizes).
Tal como num objeto da classe `list`, o operador **$** pode ser utilizado para selecionar cada uma das colunas (ou variáveis) que o constituem.
Como o `R` apenas guarda o nome do objeto que contém o *data frame* e não o seu conteúdo (as variáveis que o constituem), pode usar-se a função **`attach()`** para que o conteúdo associado ao nome das variáveis que constituem o objeto seja também guardado. Neste caso, as variáveis podem ser acedidas pelo seu nome Peng (2015).

2.2 Um outro tipo de objeto: `function`

Os objetos da classe `function` têm três componentes básicos: uma lista de argumentos, um corpo e um ambiente. A lista de argumentos, entre parêntesis curvos, é constituída por objetos separados por vírgulas. O corpo da função, entre chavetas, é constituído por uma ou mais regras separadas entre si por ponto e vírgula. O ambiente da função é o ambiente que se encontra ativo quando a função é chamada. A sintaxe específica para a definição de um objeto da classe `function` é,

```
f <- function(x,y,...){expre}
```

Também para estes objetos é possível usar coerção explícita com `as.function`.

De seguida, são abordados os conceitos de função vetorizável, predicado e funcional.

Uma função diz-se **vetorizável** se a sua avaliação num objeto é efetuada componente a componente. Todas as funções definidas em `R` são vetorizáveis. Por exemplo, aplicando a função **`sqrt()`** a um vetor, a raiz quadrada é avaliada em cada elemento do vetor. Graças à propriedade vetorizável das funções, muitos ciclos **`for`** são evitados.
Esta propriedade das funções do `R` deve ser muito bem entendida. Para o efeito é dado um exemplo simples, mas bastante elucidativo desta potencialidade.

Exemplo 2.2.1.

Supondo que se pretende calcular o vetor média móvel 3 associado a um conjunto de dados numéricos `x = x1 x2 x3 ... xn`, a ideia é construir médias sucessivas com 3 elementos de `x`:

```
(x1+x2+x3)/3, (x2+x3+x4)/3, ..., (x(n-2)+x(n-1)+xn)/3
```

Se o comprimento de `x` é n, o comprimento do vetor médias móveis 3 é $n-2$.
Para que o programa faça exatamente o que se pretende, é necessário ter em mente que as operações mandadas executar entre objetos são feitas componente a componente.
Por exemplo, se se pretender comparar cada elemento de um vetor numérico com um

outro objeto numérico, é possível fazê-lo, sem que seja necessário que a instrução correspondente percorra todos os elementos do vetor, como se pode verificar no exemplo seguinte:

Exemplo 2.2.2.

```
a<-c(8,3,6,9,4)
b<-c(6,3,4)
compare<-a<b
```

output:

```
> compare
Warning message:
In a < b : longer object length is not a multiple of shorter object length
[1] FALSE FALSE FALSE FALSE FALSE
```

Cada componente de a é comparada com cada componente de b até que tal seja possível (o R extende o objeto com menor comprimento de forma a que fique com o mesmo comprimento do outro com o qual está a operar, até o máximo possível: se não tiverem comprimentos múltiplos, imprime um aviso).
Voltando ao exemplo 2.2.1, a ideia passa por mandar efetuar a soma (e posterior divisão por 3) dos seguintes três vetores de comprimento $n-2$.

```
x1 x2 .... xn-2
x2 x3 .... xn-1
x3 x4 .... xn
```

Uma vez que a execução da soma é feita componente a componente, a intrução é do seguinte tipo:

```
x<-c(x1,x2,...,xn)
(x[1:(n-2)]+x[2:(n-1)]+x[3:n])/3
```

Uma função **predicado** é uma função cujo contradomínio é o conjunto {TRUE, FALSE}. São exemplos de funções predicado as funções do R que testam se determinado objeto pertence a determinada classe ou tipo. Todas as operações de comparação são funções predicado, e a sua escrita é sempre acompanhada pelo prefixo *is*. Algumas funções predicado em linguagem R são: **is.list()**, **is.numeric()**, **is.data.frame()**, **is.character()**, entre outras.
Finalmente, uma **funcional** é uma função que tem como argumento outra função. O R é uma linguagem de expressões: todos os comandos executados são funções ou expressões. Muitas funções em R utilizam código *C, Fortran*, etc., mas muitas outras estão em código R pelo que não diferem muito das que podem ser definidas pelo utilizador.
Existe um conjunto cada vez mais alargado de funções incorporadas em R ou disponíveis em *packages* específicos. O R é uma linguagem em constante evolução.

2.2.1 Algumas funções especiais

Nesta subsecção é feito um breve apanhado de funções úteis para a programação, começando por instruções ou funções que os programas mais clássicos dispõem.
Como qualquer linguagem, o R permite executar expressões condicionalmente:

```
1 if (expre1) expre2 else expre3
```

Com significado óbvio: Se expre1=TRUE calcula expre2
Se expre1=FALSE calcula expre3.
No entanto, existe uma versão vetorizável desta instrução que é **ifelse()**:

```
1 ifelse( vec.test, vec.se.true, vec.se.false)
```

Quando aplicada a um vetor, devolve um vetor cujo comprimento é igual ao maior comprimento do vetor do seu argumento e cujo elemento i é vec.se.true[i] se vec.test[i] se verifica e vec.se.false[i] caso contrário.
Quando existem diversas possibilidades de execução, para evitar a aplicação de muitos ifs , o R dispõe da função **switch()**.

```
1 switch(opcao, expre1,expre2,...,expren)
```

Se opcao é um número i, calcula a i-ésima expressão:

```
x<-1:3; switch(1,x+1,x+2,x+3)
[1] 2 3 4
x<-1:3;switch(2,x+1,x+2,x+3)
[1] 3 4 5
```

Se opcao é uma *string*, calcula a expressão cujo nome é essa *string*:

```
switch("a1",a1=x+1,a2=x+2,a3=x+3)
[1] 2 3 4
```

Existem também comandos para execução repetitiva em lacetes ou ciclos. Regra geral estas instruções levam à realização de muitos cálculos.
Da maneira que o R está desenhado, é possível utilizar funções vetorizáveis, como por exemplo da família **apply**, que evitam a utilização de comandos como **for**, ou **repeat**.

Exemplo 2.2.3.

Caso se pretenda utilizar a função **switch**(i,x+1,x+2,x+3) de forma repetida para os diversos valores admissíveis do primeiro parâmetro, pode ser aplicado o comando **for** da seguinte forma:

```
f<-function(i){switch(i,x+1,x+2,x+3)}
res<-list()
for (i in 1:3){res[[i]]=f(i)}
```

output:

```
> res
[[1]]
[1] 2 3 4

[[2]]
[1] 3 4 5
```

```
[[3]]
[1] 4 5 6
```

ou utilizar-se função **lapply()**:

```
f<-function(i){switch( i,x+1,x+2,x+3)}
lapply(1:3,f)
[[1]]
[1] 2 3 4

[[2]]
[1] 3 4 5

[[3]]
[1] 4 5 6
```

A função usada no Exemplo 2.2.3 é um elemento da família **apply**. De seguida são listados os elementos desta família, com uma breve explicação:

- Se `x` é um objeto da classe `matrix`, `array` ou `data.frame`, tem-se:

  apply`(X,MARGIN,FUN,...)` que aplica a função `FUN` à dimensão especificada. No caso em que `X` é um objeto da classe `matrix` ou `data.frame`, o argumento `MARGIN=1` indica as linhas e `MARGIN=2` indica as colunas.

- Se `X` é um objeto da classe `list` e se pretende aplicar `FUN` a todos os seus componentes, é utilizada a função: **lapply**`(X, FUN, ...)`.

- Para aplicar uma função `FUN` a todos os componentes de uma lista, onde se pretende obter o resultado simplificado numa estrutura de vetor, é usada a função:
  sapply`(X, FUN, ..., simplify = TRUE)`.

- Para aplicar uma função a um conjunto de valores selecionados através de uma combinação única de níveis de fatores utiliza-se a função:
  tapply`(X, INDEX, FUN = NULL,..., simplify = TRUE)` em que `x` é um objeto do tipo atómico (habitualmente um vetor), `INDEX` é uma lista de fatores, todos com o mesmo comprimento de `X` e `FUN` é, como habitualmente, a função a aplicar.

De seguida é apresentado mais um exemplo, desta vez da aplicação de **tapply()**.

```
data(PlantGrowth)
attach(PlantGrowth)
names(PlantGrowth)
[1] "weight" "group"
levels(group)
[1] "ctrl" "trt1" "trt2"
tapply(weight,group,function(x)sum(log(x)))
    ctrl     trt1     trt2
16.09814 15.26599 17.06643
```

Além desta família, existem outras funcionais que evitam a utilização de instruções do tipo *goto*, sendo muito utilizadas neste trabalho. A Tabela 2.1 lista as funções mais utilizadas:

Tabela 2.1: Funções utilizadas.

Package **base do R**	*Packages* **stringr, stringi e Prob**
outer()	**stri_sub()**
apply/sapply()	**str_trim()**
unique()	**probspace()**
sample()	**urnsamples()**
setequal()	**Prob()**
subset()	
which()	
rep()	
split()	
unlist()	
merge()	
cbind/rbind()	
paste()	
gsub()	

outer() - Trata-se de uma funcional na medida em que um dos seus argumentos é também uma função. Quando aplicada a duas sequências numéricas, é gerado um

objeto, da classe `matrix`, cuja dimensão é o comprimento da primeira sequência pelo comprimento da segunda. Mas esta função não se aplica só a vetores podendo operar com matrizes. Os seus argumentos são:

- `X`, `Y` - `vector` ou `matrix`;
- `FUN` - uma função. Por defeito está o operador produto (`*`), podendo ser também usados operadores lógicos (`<, >, =`), por exemplo.

No caso em que `X` e `Y` são vetores, a função **`outer()`** atua de maneira que, sequencialmente, cada elemento do vetor `X` é operado com cada elemento de `Y`, até esgotar os elementos de `Y`. O resultado desta operação sequencial (cíclica) é devolvido num objeto da classe `matrix`, em que, cada coluna j contém os resultados da operação do elemento j de `X` com todos os elementos de `Y`.
Considera-se o exemplo em que se pretende comparar cada elemento do vetor `x` com cada um dos elementos do vetor `y`. O resultado é uma matriz lógica, conforme se pode verificar no exemplo seguinte:

Exemplo 2.2.4.

```
x<-c(7,10,6,8,9)
y<-c(10,3,9,11,4,1)
out<-outer(X=x,Y=y,FUN="<")
```

`output:`

```
> out

      [,1]  [,2]  [,3] [,4]  [,5]  [,6]
[1,]  TRUE FALSE  TRUE TRUE FALSE FALSE
[2,] FALSE FALSE FALSE TRUE FALSE FALSE
[3,]  TRUE FALSE  TRUE TRUE FALSE FALSE
[4,]  TRUE FALSE  TRUE TRUE FALSE FALSE
[5,]  TRUE FALSE FALSE TRUE FALSE FALSE
```

Como se pode observar do *output* do Exemplo 2.2.4, a dimensão do objeto `out` é $(5,6)$, isto é, cinco linhas (comprimento do objeto `x`) e seis colunas (comprimento do objeto `y`).
Uma outra função muito utilizada neste trabalho é a função **`unique()`**.

A função **`unique()`** aplicada a um objeto `x` (`vector`, `matrix`, `array` ou `data.frame`), devolve um objeto da mesma classe e tipo, mas sem elementos duplicados. Num `array` atua em qualquer uma das dimensões. Num `data.frame` opera apenas sobre linhas.
As variáveis em objetos como `vector`, `matrix`, `array`, `data.frame`, etc. são variáveis indexadas. A função **`which()`**, cujo argumento é um objeto do tipo lógico, retorna o índice do elemento associado ao valor lógico `TRUE` dada uma condição. Esta função tem como principais argumentos:

- `x` - `vector`, `matrix` ou `data.frame`;
- `arr.ind` - condição lógica. Se `TRUE`, em objetos da classe `matrix` ou `data.frame`, retorna os índices linha coluna.

Neste trabalho, são também necessárias funções que operem sobre listas. É o caso de **`unlist()`** que permite que os componentes de um objeto da classe `list` sejam devolvidos num vetor composto por todos os elementos dessa mesma lista. Esta função tem como principais argumentos:

- `x` - lista ou vetor a decompor;
- `use.names` - condição lógica. Se `TRUE`, os nomes atribuídos aos componentes da lista são mantidos.

Funções que operam sobre um `data.frame`, tais como **`split()`**, que permite efetuar uma partição de um objeto (`vector` ou `data.frame`), em grupos através da definição de um objeto da classe `factor`. O *output* é um objeto da classe `list` e os argumentos principais da função são:

- `x` - `vector` ou `data.frame` com os elementos a serem divididos em grupos;
- `f` - fator a definir a partição pretendida.

Exemplo 2.2.5.

Caso se pretenda separar as idades de um determinado conjunto de indivíduos por género, tem-se:

```
age<-c(rep(c(18,19,20,21),c(1,3,6,2)))
gender<-rep(c('Male','Female'),c(4,8))
dtag<-data.frame(age,gender)
spgen<-split(x=age,f=gender)
```

output:

```
   age gender
1   18   Male
2   19   Male
3   19   Male
4   19   Male
5   20 Female
6   20 Female
7   20 Female
8   20 Female
9   20 Female
10  20 Female
11  21 Female
12  21 Female

> spgen
$Female
[1] 20 20 20 20 20 20 21 21
$Male
[1] 18 19 19 19
```

Ainda a função **merge()** que permite unir dois objetos da classe `data.frame`, por colunas ou linhas. Esta função tem como principais argumentos:

- `x`, `y` - dois objetos a serem unidos num só;
- `by` - especificações das colunas utilizadas para a união.

Exemplo 2.2.6.

Consideram-se os objetos df1 e df2, da classe data.frame, com informação referente à idade (*age*) e género (*gend*) de um certo conjunto de indivíduos em df1 e informação referente à idade e cidade (*city*) onde habitam de apenas parte desse conjunto de indivíduos em df2. Pretende-se unir estes dois objetos e obter a informação relativa à idade, género e cidade dos indivíduos que possuam essas três informações. Para o efeito, tem-se:

```
df1=data.frame(age=sample(18:28),gend=rep(c("Male","Female"),c
    (4,7)))
df2=data.frame(age=sample(18:28,5),city=rep(c("Lisboa","Porto"),
    c(2,3)))
me<-merge(x=df1,y=df2,by="age")
```

output:

```
> df1
   age   gend
1   25   Male
2   18   Male
3   22   Male
4   20   Male
5   26 Female
6   19 Female
7   24 Female
8   21 Female
9   28 Female
10  23 Female
11  27 Female
> df2
  age   city
1  22 Lisboa
2  26 Lisboa
3  23  Porto
```

```
4  24  Porto
5  28  Porto
> me
  age   gend   city
1  22   Male Lisboa
2  23 Female  Porto
3  24 Female  Porto
4  26 Female Lisboa
5  28 Female  Porto
```

Uma função extremamente útil é a função **`sample()`** apesar de só admitir como argumento principal um vetor ou um valor numérico. Neste caso, quando o argumento é um número, é considerado o vetor dos inteiros menores ou iguais ao número escolhido para argumento.
O objetivo é devolver amostras de dimensão especificada em `size` com ou sem reposição (controlado pelo argumento lógico `replace`) extraídas do vetor `x` que passa no primeiro argumento de **`sample()`**.
No *package* `prob` existe uma função, que será abordada mais à frente neste trabalho, que admite como argumento principal objetos da classe `data.frame`.
Existem ainda funções pensadas no âmbito de conjuntos e não de coleções de dados. Por exemplo, a função **`setequal()`** é uma função predicado que testa a "igualdade" de dois vetores de dados, vistos como conjuntos. Assim não são consideradas as eventuais repetições de elementos, nem a ordem em que estão dispostos ou até mesmo a classe. Por exemplo:

```
a<-c(2,3,4,2,2,1,5)
b<-c('5','1','2','3','4')
class(a);class(b)
[1] "numeric"
[1] "character"
setequal(a,b)
[1] TRUE
```

Evidentemente que neste caso o objeto a passou a ser character quando foi operado com b através da função **setequal()**.
A este propósito, e precisamente no mesmo âmbito, o operador %in%, usado entre dois vetores, verifica se cada elemento do primeiro vetor se encontra no segundo sem ter em conta repetições ou ordem.
Estas funções não são adequadas a coleções de dados, uma vez que nestas as repetições de elementos são importantes e a ordem com que estes surgem pode não ser irrelevante. No Capítulo 3, serão abordadas funções alternativas a estas implementadas no *package* prob.

2.2.2 Manipulação de *strings*

Por vezes há necessidade de lidar com dados de natureza não numérica como, por exemplo, nomes, letras, espaços, etc. É certo que a linguagem R pode, à partida, não ser a mais adequada para o efeito, quando comparada com as linguagens *Python*, *Perl* ou *Ruby*. No entanto, o programa R dispõe de ferramentas bastante úteis no que diz repeito à matéria em questão, disponíveis em diversos *packages*.
Existem imensos livros sobre linguagem R para uma enorme variedade de métodos estatísticos, gráficos e visualização de dados, bem como aplicações em diversas áreas, tais como ecologia, genética, psicologia, finanças , economia, etc. No entanto, é ainda escassa bibliografia no que diz respeito ao tratamento de dados de natureza não numérica, como caracteres ou conjunto/cadeia de caracteres, globalmente conhecidos como *strings* (Sanchez (2013)).
Uma razão para esta falta de recursos pode dever-se ao facto de a linguagem R não ser especificamente direcionada para a manipulação de *strings*, sendo vista, principalmente, como uma linguagem de computação e programação de dados essencialmente numéricos. Ainda assim, existe documentação muito útil em *blogs* (*stackoverflow*, *Rbloggers*,...) e em certos artigos, nomeadamente, artigos de *Hadley Wickham*.
Mesmo que não se pretenda a análise ou tratamento deste tipo de dados, existem muitas vezes caracteres em, por exemplo, nomes de linhas ou colunas, datas, valores monetários, longitude, latitude, em que existe necessidade de operar sobre esta informação. São recorrentes os casos em que é necessário, por exemplo, remover um caractere presente no nome de uma variável, ou substituir um determinado caractere dos próprios dados, ou

converter uma legenda para letras maiúsculas ou minúsculas, ou ainda, na importação de ficheiros de ambientes exteriores, modificar categorias, remover espaços, etc..
Em R, é possível lidar com este tipo de situações que envolve manipulação e tratamento de *strings*.
A função **gsub()**, disponível no *package* base, permite substituir um ou mais caracteres, presentes numa *string*, por outros à escolha. Esta função tem como principais argumentos:

- pattern - caractere ou *string* que se pretende substituir;
- replacement - caractere ou *string* escolhida para a substituição. Por defeito o caractere escolhido é o espaço em branco. '';
- - *string* ou vetor de *strings* que contém os caracteres que se pretendem a substituir.

Exemplo 2.2.7.

Considerando o objeto vchar, de classe character, se se pretender retirar os caracteres numéricos existentes e substituí-los por outro caractere à escolha, (a sintaxe utilizada para identificar os caracteres numéricos é a expressão pattern='\\d'), tem-se:

```
1 vchar<-c("1 a" ,"2 b", "3 c", "4 d", "5 e" ,"1 f")
2 nonumeric<-gsub(pattern="\\d",".",x=vchar)
```

output:

```
> nonumeric
[1] ". a" ". b" ". c" ". d" ". e" ". f"
```

Nesta subsecção são ainda aboradas funções existentes em *packages* específicas do R, que ajudam a lidar com este tipo de dados.

Exemplo 2.2.8.

Considerando, por exemplo, o objeto x e a *string* 'Marisa', pretende verificar se existe uma correspondência entre o primeiro elemento de x e a *string* 'Marisa':

```
x<-c("   Marisa "," 25 ","Mar. 25 ")
y<-"Marisa"
```

output:

```
> x==y
[1] FALSE FALSE FALSE
```

Muitas vezes, em certas coleções de dados, a informação não é inserida da melhor forma. De notar que um espaço em branco também é considerado um caractere e por vezes podem ocorrer erros de leitura devido a estes espaços em branco. No exemplo 2.2.8 é possível verificar que no objeto x, o elemento `'  Marisa '` não contém apenas os caracteres `'M'`, `'a'`, `'r'`, `'i'`, `'s'`, `'a'`, contém também alguns espaços em branco que também são considerados caracteres.
O resultado do *output* indica que a *string* `'Marisa'` não corresponde ao primeiro elemento do objeto x, isto porque `'Marisa'` e `'  Marisa '` não são a mesma *string*.

Exemplo 2.2.9.

Por outro lado, se se pretendesse verificar se a *string* `'  Marisa '` corresponde ao primeiro elemento do objeto x, ter-se ia:

```
x<-c("   Marisa "," 25 ","Mar. 25 ")
y2<-"   Marisa "
```

output:

```
> x==y2
[1]  TRUE FALSE FALSE
```

Ou seja, o primeiro elemento do objeto x e *string* y2 têm exatamente os mesmos caracteres, sendo assim considerados a mesma *string*.
Por vezes, pretende-se encontrar um certo elemento presente numa base de dados e, por não se considerarem estes espaços em branco, tal não é possível. Por este facto, é necessária especial atenção na construção de uma base dados, como por exemplo, em *excel*,

sendo de evitar a colocação de espaços em branco. No entanto, o R tem uma solução para este tipo de situações (Wickham (2010)).
O *package* stringr, disponibiliza, entre outras, a função:

- **str_trim()** - Esta função tem como finalidade eliminar os espaços em branco existentes numa determinada *string*. Estes caracteres, quase sempre indesejáveis, geralmente aparecem como os primeiros e/ou últimos caracteres (à esquerda e/ou à direita) de uma cadeia de caracteres, como no Exemplo 2.2.8. Esta função tem como argumentos:
  - string - um objeto, de classe character;
  - side= **c** ('both', 'left', 'right') - opção para remover os espaços em branco à esquerda, à direita ou ambos.

Exemplo 2.2.10.

Voltando à *string* ' Marisa ', com a função **str_trim()**, é então possível detetar os espaços em branco que possam existir e removê-los:

```
x<-c("   Marisa "," 25 ","Mar. 25 ")
trimming<-str_trim(string=x[1],side="both")
```

output:

```
> x[1]
[1] "   Marisa "

> trimming
[1] "Marisa"
```

Por defeito, esta função aplica o argumento side='both', removendo os espaços em branco à esquerda e à direita. Curiosamente, este *package* também disponibiliza a função **str_pad()**, que faz precisamente o contrário, isto é, adiciona espaços em branco a uma *string* específica. No entanto, não é utilizada neste trabalho.
Outras funções interessantes encontram-se disponíveis no *package* stringi, nomeadamente a função:

- **stri_sub()** - Esta função tem a interessante particularidade de possibilitar a extração de apenas alguns caracteres que formam uma determinada *string*, obtendo-se uma nova *string*, composta pelos caracteres extraídos da *string* inicial. Esta função tem como principais argumentos:
  - `str` - um objeto, de classe `character`;;
  - `from` e `to`, seleccionam, respectivamente, por ordem de posição dos caracteres, o início e fim da seleção de caracteres que se pretende (CRAN (2008)).

Exemplo 2.2.11.

Considerando novamente a *string* `'  Marisa '`, tem-se:

```
substr1<-stri_sub("   Marisa ",from=4,to=6)
substr2<-stri_sub("   Marisa ",from=7,to=9)
```

output:

```
> substr1
[1] "Mar"

> substr2
[1] "isa"
```

Como se pode verificar, no primeiro caso, são extraídos os caracteres da posição quatro, cinco e seis, não esquecendo que os espaços em branco também são considerados caracteres. No segundo caso, são selecionados os caracteres da posição sete, oito e nove.
É também possível inverter a ordem de selecção de caracteres, isto é, em vez de a selecção dos caracteres da *string* se efectuar da esquerda para a direita, passa a ser feita da direita para a esquerda:

Exemplo 2.2.12.

```
subinv<-stri_sub("   Marisa ",from=-7,to=-2)
```

output:

```
> subinv
[1] "Marisa"
```

No Exemplo 2.2.12, são selecionados os caracteres da posição dois até à posição sete, por ordem inversa, ou seja, o último caractere da *string* dada, que é um espaço em branco, passou a ser considerado o primeiro caractere e, consequentemente, o penúltimo caractere (`'a'`) passou a ser o segundo caractere . Já o caractere `'M'` ocupa agora a posição sete, quando anteriormente ocupava a posição quatro (contando com os três espaços em branco no início da *string*).
Existem outras funções interessantes no que diz respeito a manipulação de *strings*, no entanto, tendo em conta o objectivo principal deste trabalho, a abordagem deste tema restringe-se ao que é necessário para o desenvolvimento do mesmo.

Capítulo 3

O package `prob`

Em Probabilidade, define-se espaço amostral, Ω, como o conjunto de todos os resultados possíveis associados à realização de uma experiência aleatória. Por sua vez uma experiência aleatória é um procedimento que conduz a um resultado, que à partida não sabemos qual, mas que pertence a Ω. Estes dois conceitos são, assim, indissociáveis. Além disso, uma experiência aleatória deve poder ser repetida um número infinito de vezes nas mesmas condições.

Um espaço de probabilidade é um tripleto $(\Omega, \mathcal{A}, P)$, formado pelo conjunto Ω, por uma σ-*álgebra* $\mathcal{A}$ definida em Ω e uma por medida P de domínio $\mathcal{A}$, tal que $P(\Omega) = 1$, $\forall A \in \mathcal{A}, P(A) \geq 0$ e ainda $\forall A_1, A_2 \in \mathcal{A} | (A_1 \cap A_2) \neq 0, P(A_1 \cup A_2) = P(A_1) + P(A_2)$. Os elementos de $\mathcal{A}$ são os acontecimentos. Os acontecimentos são, assim, o argumento da função P e $P(A)$, para $A \in \mathcal{A}$, é a probabilidade da ocorrência A.

No caso em que Ω é finito ou infinito numerável, $\mathcal{A}$ pode ser o conjunto potência de A, uma vez que esta coleção contém todos os acontecimentos de Ω e satisfaz os axiomas das $\sigma-$algebras (fechada para complementares e união numerável de acontecimentos). Assim, qualquer subconjunto Ω é um acontecimento e, portanto, passível de lhe ser atribuído uma probabilidade. No caso em que Ω é finito $\#\mathcal{P}(\mathcal{A})$ é $2^{\#\Omega}$ que, ainda com uma cardinalidade com um crescimento exponencial, é numerável. Claro que no caso de Ω ser infinito, a cardinalidade de $\mathcal{P}(\mathcal{A})$ é já infinita não numerável e, portanto, a coleção de todos os acontecimentos de Ω já não coincide com o conjunto potência de Ω.

A representação de um espaço de probabilidade tem que incluir, necessariamente, os acontecimentos elementares que constituem o espaço de resultados e as respetivas pro-

babilidades.

O *package* `prob`, desenvolvido por G. J. Kernes, surgiu em 2013 com o objetivo de proporcionar ferramentas para manipulação de conceitos da Teoria de Probabilidade.

Em particular, contém a materialização dos conceitos fundamentais de espaço amostral assim como de espaço de probabilidades associados a experiências aleatórias muito utilizadas e concetualmente diferentes, tais como lançamento de moedas ou dados, extração de cartas de baralhos, jogo da roleta ou extração de bolas de urnas.

A construção dos espaços de probabilidade associados a experiências aleatórias do tipo anterior é feita através da definição de objetos da classe `data.frame` ou `list`.

No *package* `prob` existem funções como, por exemplo, **`urnsamples()`** que devolve todos os casos possíveis associados à experiência aleatória que consiste em retirar (com ou sem reposição, com ou sem ordem) um determinado número de "bolas" numeradas de uma urna. Estas "bolas" podem corresponder a linhas de um `data.frame` (bolas com algumas características associadas) ou simplesmente a entradas de uma sequência (caso mais simples – bolas com um número apenas). De facto a função pode ter como `output` um objeto `data.frame` se o argumento principal, (`urn`) for um vetor, ou um objeto `list` se este é um `data.frame`. No entanto, esta função tem uma *fraqueza*: as "bolas" na urna são sempre distinguíveis. Assim, se a urna tem elementos repetidos, por exemplo {"azul","azul", "amarelo"} correspondente a duas bolas azuis e uma amarela e se se pretender ver de quantas maneiras é possível retirar duas destas bolas, sem reposição e sem ordem, usando esta função,

```
urnsamples(c("azul","azul","amarelo"),2)
    X1       X2
1 azul     azul
2 azul amarelo
3 azul amarelo
```

o `output` não está correto. A razão para tal acontecer deve-se ao facto desta função operar sobre os índices dos elementos na "urna" – e a cada elemento correspondem, naturalmente, índices diferentes. Evidentemente que autor apresenta uma solução para este problema.

Estes e outros pormenores são discutidos neste Capítulo.

3.1 Conceitos Prévios

No package `prob` existem, essencialmente, duas formas de guardar e operar espaços de probabilidade de dimensão finita: através de um objeto `data.frame` ou através de um objeto `list`, como foi referido na introdução deste capítulo.

- O objeto da classe `data.frame` tem, além dos elementos que constituem o espaço amostra (acontecimentos elementares – linhas de um *data frame*), uma coluna com as probabilidades associadas a cada acontecimento elementar.

- O objeto da classe `list` tem dois componentes, `outcomes` e `probs`, onde `outcomes` é ainda uma lista. Este permite que os elementos do espaço amostra, (`outcomes` – acontecimentos elementares), possam ser objetos da classe `matrix`, `list` ou `data.frame`. O componente `probs` é um vetor (do mesmo comprimento que o componente `outcomes`), que associa a cada componente de `outcomes` uma probabilidade.

A definição Laplaciana de probabilidade, usada muito frequentemente em espaços de dimensão finita, necessita que os resultados elementares sejam equiprováveis pois só assim a fórmula $\frac{\#A}{\#\Omega}$ pode ser usada para o cálculo de $P(A)$. Existem, portanto, funções definidas neste *package* cuja função de probabilidade é uniforme nos n pontos que constituem o espaço amostral. São exemplos as funções **`tosscoin()`**, **`rolldie()`**, **`cards()`** ou **`rolette()`**.

As duas primeiras definem um espaço amostral correspondente ao um lançamento de uma moeda ou ao lançamento de um dado um número finito de vezes, sempre nas mesmas condições. Estas duas funções têm um argumento designado por `times`, relativo ao número de lançamentos e `makespace` que, se `TRUE` gera o espaço de probabilidades associado à realização da experiência do lançamento de uma moeda (ou dado) um número finito de vezes.

Já as funções **`cards()`** ou **`rolette()`** permitem a definição de um espaço de probabilidade associado a uma única extração ou realização da experiência, também fazendo igual a `TRUE` o argumento `makespace`.

A justificação para esta diferença essencial (não permite repetição das experiências) pode estar relacionada com a complexidade na definição de Ω. Cada carta tem um número, `rank`, e um naipe `suite` associado e cada entrada da roleta tem um número, `num`, e uma

cor, `color` associados.
Por outro lado estas funções não suportam experiências em que os acontecimentos elementares não são equiprováveis. No entanto, no *package* `prob` existem ainda funções que permitem definir espaços de probabilidade de uma forma menos rígida.

3.1.1 Espaços de probabilidade

Na definição de um espaço de probabilidade, definido o espaço amostral, o passo seguinte consiste em associar-lhe um modelo de probabilidade. Para o efeito, o `R` dispõe de algumas funções adequadas para definir um espaço de probabilidade correspondente a uma dada experiência, nomeadamente, a função **`probspace()`** que devolve um objeto da classe `ps`. Os argumentos:

- `x` - `vector`, ou outro objeto da classe `matrix`, `data.frame` ou `list`, representativo do espaço de resultados;
- `probs` - um vetor de probabilidades associadas a cada um dos resultados em `x`.

Quando os acontecimentos elementares não são equiprováveis esta função permite a construção do espaço de probabilidade desde que em `x` esteja o vetor que a cada elemento de `x` associe a respetiva probabilidade. O vetor de probabilidades é definido pelo utilizador.
Esta função atua de forma diferente à de **`iidspace()`**. Com esta é definido um espaço de probabilidade correspondente a uma sucessão de experiências independentes e identicamente distribuídas. Esta função tem os seguintes argumentos:

- `x` - um vetor com os resultados possíveis associados à realização de uma experiência aleatória;
- `ntrials` - número de vezes que se repete a experiência;
- `probs` - probabilidade de cada um dos resultados em x.

Exemplo 3.1.1.

Considerando três lançamentos consecutivos de uma moeda não equilibrada, o espaço de probabilidade obtém-se da seguinte maneira:

```
toss6<-iidspace(c("H","T"),ntrials=3,probs=c(0.6,0.4))
```

output:

```
> toss 3
  X1 X2 X3 probs
1  H  H  H 0.216
2  T  H  H 0.144
3  H  T  H 0.144
4  T  T  H 0.096
5  H  H  T 0.144
6  T  H  T 0.096
7  H  T  T 0.096
8  T  T  T 0.064
```

Veja-se agora o funcionamento da função **urnsamples()**. Esta função define o espaço amostral correspondente à experiência que consiste em retirar "bolas" de uma urna. Os argumentos são:

- `x` - `vector`, ou outro objeto da classe `matrix` ou `data.frame`, representativo da urna a partir da qual se realiza a amostragem;
- `size` - indicador da dimensão da amostra;
- `replace` e `ordered` - condições lógicas indicando, respectivamente, se a amostragem é feita com ou sem reposição e se a ordem entre amostras é ou não importante. Por defeito, estes argumentos assumem o valor lógico `FALSE`, considerando que a amostragem é feita sem reposição e sem ordenação.

De realçar que as "bolas" podem estar guardadas em diversos tipos de objetos (`vector`, `data.frame` ou `list`), podendo, assim, ser objetos de tipo diferente.
São considerados quatro tipos de amostragem:

- **Com reposição e ordenada** - Se a amostragem é feita com reposição, então é possível obter qualquer resultado em qualquer extração. Além disso, se for ordenada então é necessário ter em conta a ordem dos resultados obtidos em cada extracção;

Exemplo 3.1.2.

```
urn1<-urnsamples(1:3,size=2,replace=TRUE,ordered=TRUE)
```

output:

```
> urn1
  X1 X2
1  1  1
2  2  1
3  3  1
4  1  2
5  2  2
6  3  2
7  1  3
8  2  3
9  3  3
```

No Exemplo 3.1.2, os resultados obtidos nas linhas dois e quatro são idênticos, exceto na ordem em que estão dispostos. São considerados todos os pares possíveis de obter com os números *1*, *2* e *3*.

- **Sem reposição e ordenada** - Como a amostragem é feita sem reposição, então não é possível observar o mesmo resultado duas vezes em cada linha, reduzindo o número de resultados possíveis. A ordem em cada extracção continua a interessar;

Exemplo 3.1.3.

```
urn2<-urnsamples(1:3,size=2,replace=FALSE,ordered=TRUE)
```

output:

```
> urn2
  X1 X2
1  1  2
2  2  1
3  1  3
4  3  1
5  2  3
6  3  2
```

No exemplo 3.1.3, como seria de esperar, existem menos resultados possíveis, ou seja, menos linhas, pois a a amostragem é feita sem reposição.

- **Sem reposição e não ordenada** - Neste tipo de amostragem não é possível observar o mesmo resultado duas vezes em cada linha, mas, neste caso, a ordem dos resultados obtidos em cada extracção deixa de interessar, reduzindo ainda mais o número de resultados possíveis;

Exemplo 3.1.4.

```
urn3<-urnsamples(1:3,size=2,replace=FALSE,ordered=FALSE)
```

ou simplesmente:

```
urn3<-urnsamples(1:3,size=2)
```

output:

```
> urn3
  X1 X2
1  1  2
2  1  3
3  2  3
```

No Exemplo 3.1.4, a experiência é equivalente a retirar um par de elementos e verificar os resultados.

- **Com reposição e não ordenada** - A última possibilidade consiste em obter qualquer resultado em qualquer extração, sem se saber a ordem com que foram obtidos os resultados em cada extração.

  Exemplo 3.1.5.

```
urn4<-urnsamples(1:3,size=2,replace=TRUE,ordered=FALSE)
```

output:

```
> unr4
  X1 X2
1  1  1
2  1  2
3  1  3
4  2  2
5  2  3
6  3  3
```

No Exemplo 3.1.5, o tipo de amostragem é equivalente a baralhar três cartas de um baralho comum, colocá-las numa mesa com a face voltada para baixo, virar uma carta e ver o resultado, voltar a baralhar e repetir o processo.

3.1.2 Subconjuntos de um espaço amostral

Como já foi referido, um espaço amostral contém todos os resultados possíveis associados a uma experiência aleatória. Um acontecimento é subconjunto desse espaço. Em termos gerais os subconjumtos são definidos através de objetos lógicos. A condição ou propriedade que os elementos selecionados devem satisfazer, deve ser um predicado.
A função **`subset()`** do *package* `prob` estende a existente no *package* `base` a objetos da classe `ps`. Esta função permite considerar um subconjunto de um espaço amostral a partir de condições específicas. Os argumentos são:

- `x` - qualquer um dos quatro objetos abordados no Capítulo 2;
- `subset` - expressão lógica a indicar os elementos ou linhas a manter no subconjunto;
- `select` - Caso o objeto seja um `data.frame`, indica as colunas que se pretendem selecionar.

Exemplo 3.1.6.

Dado um baralho viciado de cinquenta e duas cartas, pretende-se obter o espaço de probabilidade associado ao acontecimento '*retirar uma carta do naipe de copas* (*Hearts*)'. Sabe-se que existem trezes cartas do naipe de copas e foi definido que a probabilidade associada a esse naipe é de 0.4. Se se tratasse de um baralho não viciado a probabilidade seria de 0.25, isto é, $\frac{13}{52}$. Para o efeito, são utilizadas as funções **`subset()`** e **`probspace()`** para extrair o subconjunto pretendido e definir as probabilidades associadas a cada naipe. Tem-se, então:

```
cards1<-probspace(cards(),probs=rep(c(0.2,0.3,0.4,0.1),rep(13,4)
   ))
subcard<-subset(cards1,suit=="Heart")
```

`output:`

```
> subcard
   rank  suit      probs
27    2 Heart 0.03076923
28    3 Heart 0.03076923
29    4 Heart 0.03076923
30    5 Heart 0.03076923
31    6 Heart 0.03076923
32    7 Heart 0.03076923
33    8 Heart 0.03076923
34    9 Heart 0.03076923
35   10 Heart 0.03076923
36    J Heart 0.03076923
```

```
37    Q Heart 0.03076923
38    K Heart 0.03076923
39    A Heart 0.03076923
```

No Exemplo 3.1.6, a probabilidade de cada carta em cada naipe, depende do naipe. A função **probspace()** funciona mesmo que a soma dos valores em probs não seja 1. Foi atribuído a cada uma das cartas do naipe de copas peso 0.4. Ao definir o objeto cards1 houve um reajuste no cálculo destas probabilidades. Cada carta de Copas tem um peso, ou probabilidade, de $P = \frac{0.4}{13}$.

intersect() - Esta função determina a interseção de dois subconjuntos A e B de um espaço de probabilidade. Num objeto da classe matrix ou data.frame, as comparações são efetuadas em cada linha e, como tal, intersect(A,B) devolve um objeto com a linhas que estão em ambos os subconjuntos A e B ($A \cap B$). Os argumentos desta função são:

- x, y - objetos do tipo vector, ou da classe data.frame, ou ps.

union() - Esta função determina a reunião de dois subconjuntos A e B de um espaço de probabilidade, devolvendo um objeto com as linhas que estão no subconjunto A ou no subconjunto B ($A \cup B$). Os argumentos são:

- x, y - objetos do tipo vector, ou da classe data.frame, ou ps.

setdiff() - Esta função determina a diferença entre dois subconjuntos A e B de um espaço de probabilidade, isto é, devolve as linhas que estão em A e não em B ($A \setminus B$). Os argumentos são:

- x, y - objetos do tipo vector, ou da classe data.frame, ou ps.

Nenhuma destas funções funciona para coleções de dados. Isto é, as repetições dos dados e a ordem não são tidas em conta. Os argumentos são vistos como conjuntos, do ponto de vista matemático e coleções de dados não são, necessariamente, conjuntos.
De seguida encontram-se exemplos da aplicação destas três funções:

Exemplo 3.1.7.

```
S<-cards()
A<-subset(S,suit=="Spade")
B<-subset(S,rank%in%6:10)
aintb<-intersect(A,B)
aub<-union(A,B)
adiffb<-setdiff(A,B)
```

output:

```
> aintb
   rank  suit
44    6 Spade
45    7 Spade
46    8 Spade
47    9 Spade
48   10 Spade

> aub
   rank    suit
5     6    Club
6     7    Club
7     8    Club
8     9    Club
9    10    Club
18    6 Diamond
19    7 Diamond
20    8 Diamond
21    9 Diamond
22   10 Diamond
31    6   Heart
32    7   Heart
33    8   Heart
34    9   Heart
35   10   Heart
40    2   Spade
```

```
41    3   Spade
42    4   Spade
43    5   Spade
44    6   Spade
45    7   Spade
46    8   Spade
47    9   Spade
48   10   Spade
49    J   Spade
50    Q   Spade
51    K   Spade
52    A   Spade

> adiffb
   rank  suit
40    2 Spade
41    3 Spade
42    4 Spade
43    5 Spade
49    J Spade
50    Q Spade
51    K Spade
52    A Spade
```

Nota: Quando o *package* `prob` é carregado, são exibidas algumas mensagens, entre as quais:

```
library(prob)
...
Attaching package: 'prob'

The following objects are masked from 'package:base':

    intersect, setdiff, union
```

Isto porque no *package* `base` do `R`, já existem métodos para as funções **`intersect()`**, **`union()`** e **`setdiff()`**. No entanto, estes métodos só são aplicáveis quando os argumentos são vetores atómicos

(do mesmo tipo, conceito abordado no Capítulo 2). Como tal, visto que a obtenção de espaços de probabilidade envolve a manipulação de objetos recursivos, nomeadamente da classe `list` e `data.frame`, há necessidade de criar métodos adequados para estes casos. Em particular, caso os argumentos sejam objetos atómicos, para evitar conflitos entre as múltiplas versões da mesma função, o *package* `prob` utiliza as versões do *package* `base` do `R`.

Existem mais extensões de funções já existentes, como por exemplo, a função **`isin()`**. Esta função permite comparar dois objetos relativamente aos elementos que o constituem, tendo em conta repetições e ordem. Assim, extende-se **`%in%`** do *package* `base`. Os argumentos são:

- `x`, `y` - vetores;
- `ordered` - condição lógica a indicar se a ordem do elementos é ou não importante.

Exemplo 3.1.8.

A diferença entre as duas funções fica mais clara neste exemplo:

```
x<-c(1,1,2,3,3,4)
y<-c(1,1,1,3,2)
y%in%x
all(y%in%x)
isin(x,y)
```

`output:`

```
[1] TRUE TRUE TRUE TRUE TRUE
[1] TRUE
[1] FALSE
```

De facto a coleção de dados em `y` não está em `x`. A função **`isin()`** é, assim, extremamente útil. Por exemplo, considerando novamente o espaço amostral obtido com **`urnsamples()`**, são dados dois exemplos, em que é aplicada a função **`isin()`**, com a particularidade de apresentar diferentes valores lógicos, `FALSE` (por defeito) ou `TRUE`, no argumento `ordered`. Os resultados obtidos são diferentes, como seria esperar:

Exemplo 3.1.9.

```
urn5=urnsamples(1:5,4)
x<-c(1,3,4,2)
suburn1<-subset(urn5,isin(urn5,x))
suburn2<-subset(urn5,isin(urn5,x,ordered=TRUE))
```

output :

```
> urn5
  X1 X2 X3 X4
1  1  2  3  4
2  1  2  3  5
3  1  2  4  5
4  1  3  4  5
5  2  3  4  5

> suburn1
  X1 X2 X3 X4
1  1  2  3  4

> suburn2
[1] X1 X2 X3 X4
<0 rows> (or 0-length row.names)
```

Como se pode verificar no Exemplo 3.1.9, caso seja considerada a ordem de extração dos elementos da urna, significa que os elementos do objeto `x`, quando comparados com cada uma das linhas do objeto `urn5`, apesar de, individualmente, aparecerem na linha *1*, a ordem em que estão dispostos os elementos não é a mesma. Como tal, a função **`isin()`** retorna o valor lógico `FALSE` em cada linha de `urn5`, não havendo assim nenhum subconjunto de `urn5` com as características consideradas. O mesmo não acontece quando se considera o argumento `ordered=FALSE` (por defeito).

3.1.3 Cálculo de probabilidades

Para o cálculo de probabilidades de acontecimentos, o *package* dispõe da função **`Prob()`**. Definido o espaço de probabilidade para calcular a probabilidade de um seu subconjunto basta somar os valores correspondentes da coluna `probs`.

Os argumentos de **`Prob()`** são:

- `x` - um espaço de probabilidade ou um subconjunto do mesmo;
- `event` - condição lógica usada para definir um subconjunto;
- `given` - objeto, da classe `data.frame` ou uma condição lógica que defina um subconjunto (Kerns (2013)).

Exemplo 3.1.10.

Considerando, novamente, a experiência em retirar uma carta de um baralho de cartas comum, denota-se o espaço de probabilidade associado à experiência como `cards2` e considera-se os subconjuntos A e B definidos da seguinte forma:

```
cards2<-cards(makespace=TRUE)
A<-subset(cards2,suit=="Heart")
B<-subset(cards2,rank%in%7:9)
PA<-Prob(A)
PB<-Prob(B)
```

`output:`

```
>PA
[1] 0.25

>PB
[1] 0.2308
```

Os argumentos `event` e `given` são aplicados a questões de probabilidade condicionada. A probabilidade condicionada de um subconjunto A, dada uma determinada ocorrência (ou evento) B é definida como:

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)}, \quad \text{se} \quad P(B) > 0$$

A título de exemplo, considerando os subconjuntos do Exemplo 3.1.10, tem-se:

Exemplo 3.1.11.

```
A<-subset(cards2,suit=="Heart")
B<-subset(cards2,rank%in%7:9)
pagb<-Prob(A,given=B)
```

output:

```
> pagb
[1] 0.25
```

Este processo, como seria de esperar, é equivalente a:

```
pagb2<-Prob(intersect(A,B))/Prob(B)
```

output:

```
> pagb2
[1] 0.25
```

3.1.4 Simulação de experiências

O *package* dispõe ainda de funções para simulação de experiências aleatórias. A função **sim()** devolve resultados relativos à simulação de uma experiência. Esta função tem como argumentos:

- `x` - espaço de probabilidade ou um subconjunto do mesmo;
- `ntrials` - número de vezes que se repete a experiência.

Tendo em conta que um espaço de probabilidade `x` contém todos os resultados possíveis de uma experiência (`outcomes`), bem como as probabilidades associadas a esses resultados (coluna `probs`), o objetivo passa por obter amostras aleatórias das linhas de `x`, de acordo com as respetivas probabilidades.

Exemplo 3.1.12.

Considerando, por exemplo, a experiência em retirar, aleatoriamente, uma carta de um baralho, tem-se:

```
S<-cards(makespace=TRUE)
samp<-sim(S,ntrials=5)
```

output:

```
> samp
  rank    suit
1    J   Heart
2    9 Diamond
3    7   Heart
4    K    Club
5    3    Club
```

Além desta, existe também a função **empirical()** que devolve as frequências relativas dos resultados simulados nas realizações sucessivas da experiência. O seu argumento é o objeto criado por aplicação da função **sim()**.

Exemplo 3.1.13.

Considerando, por exemplo, a experiência que consiste em lançar uma moeda duas vezes repetida 30000 vezes, o resultado que se obtém com a função **empirical()** é o seguinte:

```
S<-tosscoin(2,makespace=TRUE)
sims<-sim(S,ntrials=3000)
empi<-empirical(sims)
```

output:

```
> empi
  toss1 toss2     probs
1     H     H 0.2423333
2     T     H 0.2573333
3     H     T 0.2520000
4     T     T 0.2483333
```

Como se pode observar, na coluna `probs` figuram as frequências relativas (naturalmente próximas de 0.25) dos acontecimentos que constituem o espaço amostra.
As duas funções são, em conjunto, mais amigáveis, do ponto de vista do utilizador, que a função **`sample()`**, já existente no *package* `base` do `R`.
Pensando, por exemplo, nos dados `iris`, se se pretender simular realizações da experiência que consiste em retirar duas flores, aleatoriamente, e anotar todas as suas características, é possível fazê-lo muito rapidamente com **`sample()`**:

Exemplo 3.1.14.

```
set.seed(897654321)
amostra<-sample(1:150,2)
```

```
iris[amostra,]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
48          4.6         3.2          1.4         0.2  setosa
1           5.1         3.5          1.4         0.2  setosa
```

Com a função **`sim()`**:

```
set.seed(897654321)
S<-probspace(iris)
sim(S,ntrials=2)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.3         3.7          1.5         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
```

A única diferença é que a numeração das linhas foi alterada.
Claro que só no último caso se pode usar a função **`empirical()`**.

Capítulo 4

O jogo de *Poker*

Existem espaços de probabilidade (finitos) associados a experiências aleatórias mais complexas que ainda não foram pensados ou implementados no *package* `prob` e que podem ser exploradas, nomeadamente um espaço de probabilidade associado a um jogo de *poker*.

4.1 História

O *Poker* é um dos mais elegantes e famosos jogos de casino que existem na sua forma actual, existindo em várias versões: *Texas Hold'em, Omaha, Seven Card Stud, Seven Card Draw, Five Card Stud, Five Card Draw* são algumas delas. Trata-se de uma família de jogos de cartas que têm em comum a sorte, as apostas em dinheiro e em que é vencedor o jogador que possui a *mão* mais valiosa.

As referências ao jogo de *Poker* são diversas, havendo registo destes jogos desde o início do sec. XIX nos Estados Unidos da América. O jogo tornou-se de tal maneira popular que, desde 1969, existe o Campeonato Mundial de *Poker*, *World Series of Poker*, *WSOP*, nos Estados Unidos. Desde 1971 que os prémios deste campeonato são em dinheiro. Existem ainda versões *online* deste campeonato (ex: *World Championship of Online Poker*) e, em 2013, foi pela primeira vez campeão um português que arrecadou um prêmio de 20 649 dólares (Jornal Público (2013) e Wikipedia (2016)).

Existem diversos torneios ao vivo, dedicados a jogos de *Poker*, quer nos Estados Unidos quer na Europa, que atraem muito público e jogadores. As várias versões do jogo distinguem-se, fundamentalmente, pelo número de cartas que cada jogador recebe (sem

que os outros jogadores vejam), pelo número de trocas de cartas que pode efetuar ou pelo valor das apostas (Coffin (1949)).

A sorte associada ao jogo é suficiente para que a matematização deste envolva cálculo de probabilidades. A interpretação de probabilidade aqui utilizada é, naturalmente, a clássica, uma vez que se parte de um baralho com um número finito de cartas bem baralhadas de onde são selecionadas aleatoriamente, *à sorte*, parte destas cartas que constituem a *mão* (ou em inglês, *hands*) do jogador. Calcular estas probabilidades mais não é que um exercício de cálculo combinatório.

Na maior parte dos jogos de cartas, nomeadamente o jogo de *Poker*, cada jogador é detentor de um subconjunto do baralho em jogo. Esse subconjunto de cartas é designado por *mão*. O objetivo deste capítulo é mostrar como é possível chegar às probabilidades associadas a cada *mão*, usando `R`.

Regra geral, o baralho é de cinquenta e duas cartas, distribuídas por quatro naipes, *Copas (hearts), Espadas (spades), Ouros (diamonds)* e *Paus (clubs)*, com treze cartas cada. No *Poker* todos os naipes têm o mesmo valor e, dentro de cada naipe, há uma ordenação nas cartas, sendo o *Ás* a carta de valor mais elevado e o *duque* (dois) o de valor mais baixo. Tem-se a seguinte ordenação por ordem crescente dentro de cada naipe: *2, 3, 4, 5, 6,7, 8, 9, 10, Valete (Jack - J)*, *Dama (Queen - Q)*, *Rei (King - K)* e *Ás (Ace - A)*, sendo atribuídos os valores numéricos *11*, *12*, *13* e *14* a *J*, *Q*, *K* e *A*, como se pode observar na Figura 4.1:

Figura 4.1: Ordenação do valor numérico das cartas dentro de cada naipe.

Em algumas versões do jogo, o *Ás* pode assumir o valor um e/ou catorze. As *mãos* e a respetiva valoração são idênticas em todas as versões.

4.2 Versão *Five Draw Poker*

A regra geral num jogo de *poker* é: *O jogador que tiver melhor mão vence o jogo*. As diversas versões diferem também quanto ao valor do *Ás*, sendo que, na maioria da versões do jogo, é possível obter uma *mão* correspondente a uma sequência de cinco cartas onde o *Ás* pode assumir dois valores distintos, como por exemplo:

- *Ás, 2, 3, 4, 5* onde o *Ás* assume o valor numérico um;
- *10, Valete, Dama, Rei* e *Ás*. Neste caso, o *Ás* assume o valor numérico catorze.

De facto, como foi dito anteriormente, o *Ás* pode assumir o valor numérico *1* ou *14*. De salientar que ambas as sequências têm a mesma probabilidade de ocorrer.
Atualmente, a versão *Texas Hold'em* é considerada a modalidade de *Poker* mais popular do mundo, no entanto, a versão *Five-Card Draw*, apesar de ter perdido algum protagonismo face à versão *Texas Hold'em*, continua a ser uma variante muito jogada, sendo considerada a mais antiga do jogo.
Neste trabalho é abordada a versão *Five-Card Draw*. As regras deste jogo estão descritas a seguir:

- Antes do início do jogo, os jogadores fazem uma aposta inicial para dar consistência ao somatório das apostas (em inglês *pot*);
- Escolha do *dealer* - A partida inicia-se com cada jogador a receber uma carta do baralho de face voltada para cima. O jogador ao qual for atribuída a carta de maior valor é o *dealer*, responsável por distribuir as cartas, sendo o último a jogar (ocupa a última posição na mesa). O *dealer* distribui cinco cartas, de face voltada para baixo, a cada jogador, começando pelo jogador que se encontra à sua esquerda;
- Na mesma ordem, depois de cada jogador ter examinado a sua *mão*, começa a primeira ronda de apostas. Após a primeira ronda, os jogadores, ainda em jogo, têm a oportunidade de solicitar a troca de alguma ou todas as cartas da sua *mão*, ou não substituir nenhuma carta;
- Depois de todos os jogadores terem tido a oportunidade de trocar cartas, ocorre a última ronda de apostas e, por fim, os jogadores mostram as suas cartas que possuem aos adversários, e a melhor mão ganha o *pot*;

- Se, no final da segunda ronda de apostas, apenas um jogador permanecer no jogo, porque todo os outros não apostaram, este não é obrigado a mostrar a sua *mão* aos adversários, ganhando essa ronda (jogapoker (2009)).

4.2.1 Classificação das *Mãos*

À semelhança de outras versões, no *Five Card Draw*, existem dez *mãos* possíveis de obter, com uma combinação máxima de cinco cartas. De seguida encontram-se descritas, por ordem decrescente de valoração, cada uma dessas *mãos*, bem como uma tabela (*Tabela 4.1*) com cada uma das probabilidades associadas.
Tem-se, então:

- ***Royal flush*** - Sequência numérica mais alta com cartas do mesmo naipe, em que o *Ás* assume o valor numérico *14*, designado *Ás* alto. Se mais do que um jogador apresentar uma *mão* desta classe, não se aplica critério de desempate, pois todas as mãos desta classe têm igual valor numérico e os quatro naipes têm igual valor, como já foi referido anteriormente. Assim, dá-se o empate e o *pot* é repartido pelos jogadores empatados.

Tabela 4.1: *Royal flush*.

Mão	**Probabilidade**	**Exemplo**
Royal Flush	$P_{RF} = \frac{4}{\binom{52}{5}} = 1.539 \times 10^{-6}$	 Figura 4.2: *Royal flush* de copas.

onde:

  - $\binom{52}{5}$ são todas as combinações possíveis, ao retirar, aleatoriamente, cartas do baralho;
  - 4 é o número de sequências com *Ás alto*, em cinco cartas do mesmo naipe.

- ***Straight flush*** - Sequência ordenada de cinco cartas do mesmo naipe, em que o *Ás* pode assumir o valor numérico *1* para formar a sequência mais baixa de todas as existentes. Se mais do que um jogador apresentar uma *mão* desta classe, aquele que tiver a sequência com cartas de maior valor numérico ganha.

Tabela 4.2: *Straight flush*.

Mão	**Probabilidade**	**Exemplo**
Straight Flush	$P_{SF} = \frac{4\times10-4}{\binom{52}{5}} = 1.385\times10^{-5}$	 Figura 4.3: *Straight flush* de espadas.

onde:

  - 4×10 é o total de sequências possíveis de obter com cinco cartas do mesmo naipe;
  - 4 é o número de sequências com cartas do mesmo naipe que não pertencem a esta *mão* (os casos favoráveis a um *Royal flush*).

- ***Four of a kind*** - Quatro cartas de igual valor numérico. Se mais do que um jogador tiver uma *mão* desta classe, ganha aquele que apresentar as cartas de maior valor numérico.

Tabela 4.3: *Four of a kind*.

Mão	**Probabilidade**	**Exemplo**
Four of a kind	$P_{FK} = \frac{\binom{4}{4}\times 13\times\binom{4}{1}\times 12}{\binom{52}{5}} = 0.00024$	 Figura 4.4: *Four of a kind* de dez.

onde:

- 13 são todas as combinações possíveis de fazer com quatro cartas com o mesmo valor numérico;
- $\binom{4}{1} \times 12$ representam as combinações possíveis de fazer com as restantes cartas, para obter a última carta que compõe a esta *mão*.

- ***Full House*** - Uma *mão* com um par e um trio (ou terno). Se mais do que um jogador tiver uma *mão* desta classe, aquele que apresentar o trio com cartas de maior valor numérico vence.

Tabela 4.4: *Full house*.

Mão	**Probabilidade**	**Exemplo**
Full house	$P_{FH} = \frac{\binom{4}{3}\times 13\times\binom{4}{2}\times 12}{\binom{52}{5}} = 0.0014$	 Figura 4.5: *Full house* de sete e valete.

onde:

- $\binom{4}{3} \times 13$ são todos os trios possíveis de obter;

  - $\binom{4}{2} \times 12$ são todas as combinações possíveis de fazer com as restantes cartas, para obter um par.

- ***Flush*** - Cinco cartas do mesmo naipe, que não formem uma sequência ordenada. Se mais do que um jogador possuir uma *mão* desta classe, ganha o que tiver a carta de maior valor numérico, entre todas as cartas dos jogadores que obtiveram um *flush*. onde:

Tabela 4.5: *Flush*.

Mão	**Probabilidade**	**Exemplo**
Flush	$P_{FL} = \frac{\binom{4}{1} \times \binom{13}{5} - 4 \times 10}{\binom{52}{5}} = 0.00197$	 Figura 4.6: *Flush* de copas.

  - $\binom{4}{1} \times \binom{13}{5}$ são todas as combinações possíveis de fazer, para obter cinco cartas do mesmo naipe e, consequentemente, de diferente valor numérico,;
  - 4×10 representam a exclusão das sequências do mesmo naipe - *Straigth flush* e *Royal flush*, respectivamente.

- ***Straight*** - Sequência numérica de cinco cartas com pelo menos um naipe diferente dos restantes. À semelhança do *Straight flush*, se mais do que um jogador apresentar uma *mão* desta classe, aquele que tiver a sequência de maior valor numérico ganha. Neste tipo de *mão*, o *Ás* pode assumir o valor numérico *1* ou *14*, consoante a sequência que for possível formar.

Tabela 4.6: *Straight*.

Mão	**Probabilidade**	**Exemplo**
Straight	$P_{ST} = \frac{10\times\binom{4}{1}^5-36-4}{\binom{52}{5}} = 0.00392$	 Figura 4.7: *Straights* de *Ás*.

onde:

- $10 \times \binom{4}{1}^5$ são as sequências possíveis de obter com cinco cartas;
- 36 e 4 representam as sequências do mesmo naipe a excluir (*Straigth flush* e *Royal flush*, respectivamente).

- ***Three of a Kind*** - três cartas de igual valor numérico. Novamente, se mais do que um jogador possuir uma *mão* desta classe, ganha o que tiver o trio com cartas de maior valor numérico. Não sendo possível dois jogadores obterem exatamente o mesmo trio, devido ao facto de haver um máximo de quatro cartas de igual valor numérico, este é o único critério de desempante aplicado.

Tabela 4.7: *Three of a kind*.

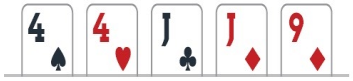
Mão	**Probabilidade**	**Exemplo**
Three of a kind	$P_{TK} = \frac{\binom{4}{3}\times 13\times\binom{12}{2}\times\binom{4}{1}^2}{\binom{52}{5}} = 0.0211$	 Figura 4.8: *Three of a kind* de noves.

onde:

- $\binom{4}{3} \times 13$ são todos os trios possíveis de obter;
- $\binom{12}{2} \times \binom{4}{1}^2$ representam todas as combinações possíveis de fazer com as restantes cartas, para obter duas cartas, de diferente valor numérico entre si e do trio.

- ***Two pairs*** - Duas cartas de igual valor numérico, duas a duas. Se mais do que um jogador apresentar uma *mão* desta classe, aquele que tiver um dos pares de maior valor numérico ganha. Se mais do que um jogador apresentar os mesmos dois pares, a 5ª carta restante de maior valor numérico de cada jogador determina o vencedor.

Tabela 4.8: *Two pairs*.

Mão	**Probabilidade**	**Exemplo**
Two pairs	$P_{TP} = \frac{\binom{4}{2}^2 \times \binom{13}{2} \times \binom{4}{1} \times 11}{\binom{52}{5}} = 0.0475$	 Figura 4.9: *Two pairs* de quadra e valete.

onde:

- $\binom{4}{2}^2 \times \binom{13}{2}$ são todas as combinações possíveis de fazer, para obter dois pares;
- $\binom{4}{1} \times 11$ representam todas as combinações possíveis de fazer, para obter a quinta carta que compõe esta mão, com valor numérico diferente dos dois pares.

- ***One pair*** -Duas cartas de igual valor numérico, tendo as restantes três cartas diferente valor numérico. Se mais do que um jogador tiver uma *mão* desta classe, aquele que possuir o par de maior valor numérico ganha. No caso de dois jogadores apresentarem exactamente o mesmo par, o critério de desempate recai a favor do jogador que possuir a carta de maior valor numérico de entre as restantes três cartas de cada um dos jogadores.

Tabela 4.9: *One pair*.

Mão	Probabilidade	Exemplo
One pair	$P_{OP} = \frac{\binom{4}{2} \times 13 \times \binom{12}{3} \times \binom{4}{1}^3}{\binom{52}{5}} = 0.4226$	 Figura 4.10: *One pair* de sena.

onde:

- $\binom{4}{2} \times 13$ são todas as combinações possíveis de fazer, de modo a obter um par;
- $\binom{12}{3} \times \binom{4}{1}^3$ representam as combinações possíveis de fazer, para obter as três últimas cartas, de diferente valor numérico entre si e do par.

- ***High card***- Por último, tem-se o caso de não sair nenhuma das *mãos* anteriores. Caso nenhum dos jogadores apresente uma das mão descritas acima, para efeito de desempate, ganha quem tiver a carta de maior valor numérico.

Tabela 4.10: *High card*.

Mão	Probabilidade	Exemplo
High card	$P_{HC} = \frac{[\binom{13}{5} - 10] \times (4^5 - 4)}{\binom{52}{5}} = 0.501$	 Figura 4.11: *High card* de *Ás*.

em que:

- $\binom{13}{5}$ são todas as combinações possíveis de fazer, para obter cinco cartas, de diferente valor numérico entre si;

- 10 são os *Straights* a excluir;
- 4^5 e 4 representam todas as combinações possíveis de fazer com os naipes, excepto os casos em que as cinco cartas são todas do mesmo naipe (Ramsey (2005)).

4.3 Implementação do jogo em R

Conhecidas as probabilidades associadas a cada uma das *mãos* do jogo em estudo, o próximo passo consiste em implementar o jogo na linguagem R. Recorrendo a funções existentes no *package* prob, a funções já implementadas em R, nomeadamente as que dizem respeito à família **apply()** e à manipulação e tratamento de *strings*, é possivel criar o espaço de probabilidade associado ao jogo em estudo.
A experiência aleatória *E* consiste em retirar, simultaneamente, cinco cartas de um baralho usual de cinquenta e duas cartas. Para representar um baralho usual de cinquenta e duas cartas, pode ser aplicado o seguinte procedimento:

- Cada carta tem um valor numérico associado (de 2 a 14);
- Cada carta tem uma letra associada (*C, D, H, S*);
- Cada uma das cartas identificadas com o valor numérico de 11 a 14, tem ainda associada uma letra (*J, Q, K, A*);
- A funcional **outer()** é utilizada para, juntamente com a função **paste()**, organizar numa matriz com quatro colunas e treze linhas, os elementos que constituem o baralho de cinquenta e duas cartas, como se pode verificar de seguida:

```
1 deck<-outer(c(2:10,"11 J","12 Q","13 K","14 A"),c("C","D","H","S
    "),"paste")
```

output:

```
> deck
      [,1]     [,2]     [,3]     [,4]
 [1,] "2 C"    "2 D"    "2 H"    "2 S"
 [2,] "3 C"    "3 D"    "3 H"    "3 S"
 [3,] "4 C"    "4 D"    "4 H"    "4 S"
 [4,] "5 C"    "5 D"    "5 H"    "5 S"
 [5,] "6 C"    "6 D"    "6 H"    "6 S"
 [6,] "7 C"    "7 D"    "7 H"    "7 S"
 [7,] "8 C"    "8 D"    "8 H"    "8 S"
 [8,] "9 C"    "9 D"    "9 H"    "9 S"
 [9,] "10 C"   "10 D"   "10 H"   "10 S"
[10,] "11 J C" "11 J D" "11 J H" "11 J S"
[11,] "12 Q C" "12 Q D" "12 Q H" "12 Q S"
[12,] "13 K C" "13 K D" "13 K H" "13 K S"
[13,] "14 A C" "14 A D" "14 A H" "14 A S"
```

Definido o baralho, é então possível construir o espaço de probabilidade associado à experiência aleatória atrás definida. Trata-se uma amostragem sem reposição e não ordenada.

O espaço de probabilidade associado a esta experiência pode então ser obtido recorrendo à função **`urnsamples()`**, do *package* `prob`. Esta função define, neste caso, um objeto da classe *data.frame*, com todos os resultados possíveis do jogo, bem como a probabilidade associada a cada um destes acontecimentos elementares.

Assim, se se quiser obter todos os resultados possíveis associados a esta experiência, basta considerar a função **`urnsamples()`** com os seguintes argumentos:

- `x=deck`;
- `size=5`
- `replace=FALSE`, `ordered=FALSE` ;

Obtém-se assim um objeto, da classe `data.frame`, em que as cinco colunas correspondem ao valor e naipe de cada carta. O número de linhas deste objeto é precisamente o número de casos possiveis do jogo, $\binom{52}{5} = 2598960$.

A este objeto, que contém todos os casos possíveis associadas ao jogo, em que cada linha é um caso possível, atribuem-se denominações adequadas às colunas geradas. Para o efeito é utilizada a à função **colnames()**. Assim, recorrendo à função **rep()**, que permite replicar os valores de um vetor, e novamente à função **paste()**, é possível criar o objeto hand0, da classe data.frame:

```
hand0<-urnsamples(x=deck,size=5)
colnames(hand0)<-paste(rep(x="card",times=5),1:5)
```

output:

```
> head(hand0)
  card 1 card 2 card 3 card 4 card 5
1    2 C    3 C    4 C    5 C    6 C
2    2 C    3 C    4 C    5 C    7 C
3    2 C    3 C    4 C    5 C    8 C
4    2 C    3 C    4 C    5 C    9 C
5    2 C    3 C    4 C    5 C   10 C
6    2 C    3 C    4 C    5 C 11 J C
```

No objeto hand0, uma vez que nem sempre há necessidade de utilizar os valores numéricos 11, 12, 13 e 14, associados a *J, D, K* e *A*, a dada altura é conveniente retirá-los. Para tal, são utilizadas as funções **stri_trim()** e **stri_sub()**, constantes do *package* stringi e stringr.

Assim, com a função **stri_sub()**, é possível extrair um número específico de caracteres presentes no objeto hand0. Por exemplo, a *string* '11 J C' possui seis caracteres, nomeadamente, os caracteres '1', '1', 'J', 'C' e dois espaços em branco '' e ''. Como já foi referido anteriormente, os caracters '1' e '1', apenas servem para associar um valor numérico a 'J'.

Se se considerar o argumento from=-4, a selecção de caracteres tem início no fim da *string* (da direita para esquerda), onde os caracteres numéricos '11', '12', '13' e '14' associados aos valores das cartas *J, Q, K* e *A* deixam de aparecer no *output*. De forma bastante simples, o argumento from=-4 impõe que, no *output*, apareçam apenas os quatro

últimos caracteres de cada *string*. De realçar, novamente, que a *string* `'11 J C'`, tem seis caracteres e, com a imposição considerada, no *output* aparecem apenas os caracteres `' J C'`. No caso da *string* `'2 C'`, esta tem apenas três caracteres, não sofrendo assim, qualquer alteração no *output*.
A título de exemplo, considerando o baralho `deck`, pretende-se extrair os quatro últimos caracteres de cada um dos elementos deste objeto. Como se trata de um objeto da classe `matrix`, a função **`apply()`** pode ser utilizada para que a função **`stri_sub()`** seja aplicada a cada uma das quatro colunas. Tem-se agora:

```
deck1<-apply(deck,2,stri_sub,from=-4)
```

`output:`

```
> deck1
      [,1]   [,2]   [,3]   [,4]
 [1,] "2 C"  "2 D"  "2 H"  "2 S"
 [2,] "3 C"  "3 D"  "3 H"  "3 S"
 [3,] "4 C"  "4 D"  "4 H"  "4 S"
 [4,] "5 C"  "5 D"  "5 H"  "5 S"
 [5,] "6 C"  "6 D"  "6 H"  "6 S"
 [6,] "7 C"  "7 D"  "7 H"  "7 S"
 [7,] "8 C"  "8 D"  "8 H"  "8 S"
 [8,] "9 C"  "9 D"  "9 H"  "9 S"
 [9,] "10 C" "10 D" "10 H" "10 S"
[10,] " J C" " J D" " J H" " J S"
[11,] " Q C" " Q D" " Q H" " Q S"
[12,] " K C" " K D" " K H" " K S"
[13,] " A C" " A D" " A H" " A S"
```

De seguida, e porque o número de caracteres de cada *string* (elementos do baralho) não é sempre igual, é necessário remover os espaços em branco, presentes nas *strings* com maior número de caracteres. Para o efeito, recorre-se à função **`str_trim()`**:

```
deck2<-apply(deck1,2,str_trim,side="left")
```

output:

```
> deck2
      [,1]   [,2]   [,3]   [,4]
 [1,] "2 C"  "2 D"  "2 H"  "2 S"
 [2,] "3 C"  "3 D"  "3 H"  "3 S"
 [3,] "4 C"  "4 D"  "4 H"  "4 S"
 [4,] "5 C"  "5 D"  "5 H"  "5 S"
 [5,] "6 C"  "6 D"  "6 H"  "6 S"
 [6,] "7 C"  "7 D"  "7 H"  "7 S"
 [7,] "8 C"  "8 D"  "8 H"  "8 S"
 [8,] "9 C"  "9 D"  "9 H"  "9 S"
 [9,] "10 C" "10 D" "10 H" "10 S"
[10,] "J C"  "J D"  "J H"  "J S"
[11,] "Q C"  "Q D"  "Q H"  "Q S"
[12,] "K C"  "K D"  "K H"  "K S"
[13,] "A C"  "A D"  "A H"  "A S"
```

Voltando ao objeto `hand0`, é possível aplicar o mesmo processo e, utilizando o objeto resultante (da mesma classe que `deck2`) como argumento da função **`probspace()`**, é possível obter o espaço de probabilidade associado à experiência em análise. É então gerado um novo objeto (`hand`), da classe `data.frame`:

```
1 h1<-apply(hand0,2,stri_sub,-4)
2 h2<-apply(h1,2,str_trim,side="left")
3 hand<-probspace(h2)
```

output:

```
> head(hand)
  card.1 card.2 card.3 card.4 card.5    probs
1    2 C    3 C    4 C    5 C    6 C 3.85e-07
2    2 C    3 C    4 C    5 C    7 C 3.85e-07
3    2 C    3 C    4 C    5 C    8 C 3.85e-07
```

```
4    2 C    3 C    4 C    5 C    9 C 3.85e-07
5    2 C    3 C    4 C    5 C   10 C 3.85e-07
6    2 C    3 C    4 C    5 C    J C 3.85e-07
```

Como se pode observar, a última coluna do objeto `hand`, indica a probabilidade associada a cada um dos acontecimentos elementares (cada uma das linhas) que, naturalmente, é a mesma para todos os casos.

4.3.1 Partição do Espaço de Probabilidade

As dez *mãos* existentes no jogo são acontecimentos mutuamente exclusivos e exaustivos do objeto `hand`, constituindo uma partição deste espaço.
Como o espaço é finito, é possível construir exaustivamente (e em extensão) estes acontecimentos. Para tal, deve ser definida uma propriedade que os identifique de maneira única e sem qualquer ambiguidade, e que seja compreendida pelo programa.
Uma forma de definir estes acontecimentos consiste filtrar os casos correspondentes a cada uma das dez *mãos* existentes no jogo.
Começando com as *mãos Pair, Two pairs, Three of a kind, Full house* e *Four of a kind*, a valoração atribuída a cada uma destas *mãos* depende apenas do valor numérico de cada umas cinco cartas. Assim, o primeiro passo, para definir os acontecimentos associados, consiste em criar o objeto `handnum`, da classe `data.frame`, onde são considerados apenas os caracteres numéricos de cada um dos elementos do objeto `hand0`. Para tal, recorre-se, novamente, às funções **`apply()`** e **`stri_sub()`**.
Para o efeito, a função **`stri_sub()`** assume os argumentos `from=1` e `to=2`, onde são selecionados os dois primeiros caracteres de cada um dos elementos do objeto `hand0`. Assim, para todas as *mãos* em que é necessário considerar o valor numérico de cada uma das cinco cartas, é mais adequado utilizar este novo objeto, `handnum`. De notar que é também utilizada a função **`as.numeric()`** em cada uma das colunas do novo objeto criado, para que os seus elementos sejam lidos como valores numéricos e não como *strings*:

```
1 hd<-apply(hand0,2,stri_sub,1,2)
```

```
2 handnum<-as.data.frame(apply(hd,2,as.numeric))
```

output:

```
> head(handnum)
  card 1 card 2 card 3 card 4 card 5
1      2      3      4      5      6
2      2      3      4      5      7
3      2      3      4      5      8
4      2      3      4      5      9
5      2      3      4      5     10
6      2      3      4      5     11
```

No que diz respeito às *mãos Pair, Two pairs, Three of a kind, Full house* e *Four of a kind*, os casos favoráveis associados a cada um deste acontecimentos podem ser obtidos de uma forma simples e intuitiva. Tendo em conta que não é possível obter, em simultâneo, cartas do mesmo naipe e de igual valor numérico, então, para estas mãos, não há necessidade de verificar se as cartas são ou não todas do mesmo naipe. Se sair um par, por exemplo, essas duas cartas já são de naipes diferentes.

Assim, para obter todos os casos favoráveis à *mão Pair*, inicialmente, são percorridas todas as linhas do objeto `handnum` e, recorrendo à função **`unique()`**, obtém-se um objeto da classe `list` em que cada um dos seus componentes é um objeto numérico, correspondente a cada uma das linhas do objeto `handnum`, mas sem valores repetidos (`uni`). Ou seja, no caso de uma linha constituída pelos valores numéricos `2 2 3 4 5`, apenas são listados os valores `2 3 4 5`. Posteriormente, obtém-se o comprimento (`len`) de cada um dos componentes (vetores) do objeto `uni`. Facilmente se percebe que os componentes do objeto `uni` com comprimento cinco correspondem às linhas do objeto `handnum`, constituídas por cinco valores numéricos distintos. Por outro lado, os componentes de `uni` com comprimento quatro, correspondem agora às linhas de `handnum` em que dois dos cinco elementos que as constituem, têm igual valor numérico.

Do ojeto gerado (`Pair`), da classe `data.frame`, é possível observar, por exemplo, um par de *duques* ou de *ternos*, com as restantes três cartas diferentes entre si:

```
uni<-apply(handnum,1,unique)
len<-sapply(uni,length)
Pair<-subset(x=hand,len==4)
```

output:

```
> head(Pair)
   card.1 card.2 card.3 card.4 card.5    probs
10    2 C    3 C    4 C    5 C    2 D 3.85e-07
11    2 C    3 C    4 C    5 C    3 D 3.85e-07
12    2 C    3 C    4 C    5 C    4 D 3.85e-07
13    2 C    3 C    4 C    5 C    5 D 3.85e-07
23    2 C    3 C    4 C    5 C    2 H 3.85e-07
24    2 C    3 C    4 C    5 C    3 H 3.85e-07
```

No caso de sairem as mãos *Two pairs*, *Three of a kind*, *Full house* ou *Four of a kind*, o processo é semelhante, mas com algumas modificações.
Se se pretender obter os casos favoráveis aos acontecimentos *Two pairs* e *Three of a kind*, basta considerar os componentes da lista `uni` que tenham comprimento três (`len3`), correspondentes às linhas do objeto `handnum` em que dois ou três dos cinco elementos que a constituem, apresentam o mesmo valor numérico:

```
len3<-subset(handnum,len==3)
```

output:

```
> head(len3)
    card 1 card 2 card 3 card 4 card 5
436      2      3      4      2      3
437      2      3      4      2      4
448      2      3      4      2      2
449      2      3      4      2      3
450      2      3      4      2      4
```

```
461      2      3      4      2      2
```

Do subconjunto `len3`, é possível encontrar os seguintes casos:

- quatro elementos com o mesmo valor numérico dois a dois, como se pode verificar na linha `436`. Esta apresenta os elementos `2 3 4 2 3`;
- três e apenas três elementos com o mesmo valor numérico , como é o caso da linha `461` que é constituída pelos elementos `2 3 4 2 2`.

Facilmente se percebe que, o primeiro exemplo corresponde um dos casos favoráveis ao acontecimento: *Two pairs*. Assim, para obter todos estes casos, basta considerar as linhas em que o número máximo de elementos com o mesmo valor numérico (`lenmax`) é dois. Para que os casos encontrados se associem às linhas do objeto `hand`, recorre-se à função **`rownames()`**, sendo gerado o objeto `Twopairs`, da classe `data.frame`:

```
1 tab<-apply(len3,1,table)
2 lenmax<-apply(tab,2,max)
3 Two_pairs<-hand[rownames(subset(len3,lenmax==2)),]
```

`output:`

```
> head(Two_pairs)
    card.1 card.2 card.3 card.4 card.5    probs
436    2 C    3 C    4 C    2 D    3 D 3.85e-07
437    2 C    3 C    4 C    2 D    4 D 3.85e-07
449    2 C    3 C    4 C    2 D    3 H 3.85e-07
450    2 C    3 C    4 C    2 D    4 H 3.85e-07
462    2 C    3 C    4 C    2 D    3 S 3.85e-07
463    2 C    3 C    4 C    2 D    4 S 3.85e-07
```

O segundo exemplo corresponde a um dos casos favoráveis ao acontecimento: *Three of a kind*. De forma análoga ao caso anterior, para obter todos estes casos, consideraram-se agora as linhas em que o número máximo de elementos com o mesmo valor numérico é

três, sendo gerado o objeto `Three_kind`, da mesma classe que o objeto gerado no caso anterior:

```
Three_kind<-mao[rownames(subset(len3,lenmax==3)),]
```

output :

```
> head(Three_kind)
    card.1 card.2 card.3 card.4 card.5    probs
448    2 C    3 C    4 C    2 D    2 H 3.85e-07
461    2 C    3 C    4 C    2 D    2 S 3.85e-07
486    2 C    3 C    4 C    3 D    3 H 3.85e-07
499    2 C    3 C    4 C    3 D    3 S 3.85e-07
523    2 C    3 C    4 C    4 D    4 H 3.85e-07
536    2 C    3 C    4 C    4 D    4 S 3.85e-07
```

Seguindo o mesmo raciocínio, para se obter os casos favoráveis aos acontecimentos *Full house* e *Four of a kind*, basta considerar os componentes do objeto `uni` que tenham comprimento dois (`lenfull`), correspondentes às linhas de `handnum`, onde apenas dois dos cinco elementos que as constituem apresentam valores distintos:

```
lenfull<-subset(handnum,len==2)
```

output :

```
> head(lenfull)
      card 1 card 2 card 3 card 4 card 5
10473      2      3      2      3      2
10474      2      3      2      3      3
10486      2      3      2      3      2
10487      2      3      2      3      3
10840      2      3      2      2      3
10852      2      3      2      2      2
```

Do subconjunto `lenfull`, é possível encontrar os seguintes casos:

- três elementos com o mesmo valor numérico e os restantes dois elementos também de igual valor numérico, como é o caso da linha `10473`, constituída pelos elementos `2 3 2 3 2`;
- quatro elementos com o mesmo valor numérico, como se pode verificar na linha `10852`, contendo os elementos `2 3 2 2 2`.

Facilmente se percebe que o primeiro exemplo corresponde a um dos casos favoráveis ao acontecimento: *Full house*. Para obter todos estes casos, basta considerar as linhas em que o número máximo de elementos com o mesmo valor numérico (`maxfull`) é três. Obtém-se assim o objeto `Full_house`, da classe `data.frame`:

```
tabfull<-apply(lenfull,1,table)
maxfull<-apply(tabfull,2,max)
Full_house<-mao[rownames(subset(lenfull2,maxfull==3)),]
```

output:

```
> head(Full_house)
      card.1 card.2 card.3 card.4 card.5    probs
10473    2 C    3 C    2 D    3 D    2 H 3.85e-07
10474    2 C    3 C    2 D    3 D    3 H 3.85e-07
10486    2 C    3 C    2 D    3 D    2 S 3.85e-07
10487    2 C    3 C    2 D    3 D    3 S 3.85e-07
10840    2 C    3 C    2 D    2 H    3 H 3.85e-07
10853    2 C    3 C    2 D    2 H    3 S 3.85e-07
```

O segundo exemplo correponde a um dos casos favoráveis ao acontecimento: *Four of a kind*. Para estes casos são consideradas as linhas em que o número máximo de elementos com o mesmo valor numérico (`maxfull`) é quatro, sendo gerado o objeto `Four_kind`, da classe `data.frame`:

```
Four_kind<-hand[rownames(subset(lenfull,maxfull==4)),]
```

output:

```
> head(Four_kind)
      card.1 card.2 card.3 card.4 card.5    probs
10852    2 C    3 C    2 D    2 H    2 S 3.85e-07
11543    2 C    3 C    3 D    3 H    3 S 3.85e-07
29276    2 C    4 C    2 D    2 H    2 S 3.85e-07
30621    2 C    4 C    4 D    4 H    4 S 3.85e-07
46572    2 C    5 C    2 D    2 H    2 S 3.85e-07
48535    2 C    5 C    5 D    5 H    5 S 3.85e-07
```

De notar que existem outros formas de obter os mesmos resultados, sendo que quanto mais simples e optimizados forem os processos utilizados, melhor.
De seguida, é analisado o procedimento para quando também o naipe das cartas é tido em consideração. Esta situação aplica-se às *mãos Flush, Straight flush, Royal flush* e *Straight*. Em relação às três primeiras *mãos*, estes acontecimentos são constituídos por cinco cartas do mesmo naipe. Para tal, uma forma simples de obter estes casos consiste em criar um novo objeto (`handsuit`), da classe `data.frame` onde são considerados apenas os caracteres referentes ao naipe de cada um dos elementos de `hand0`:

```
handsuit<-as.data.frame(apply(hand0,2,stri_sub,-1,6))
```

output:

```
> head(handsuit)
  card 1 card 2 card 3 card 4 card 5
1      C      C      C      C      C
2      C      C      C      C      C
3      C      C      C      C      C
4      C      C      C      C      C
5      C      C      C      C      C
6      C      C      C      C      C
```

Mais uma vez, são percorridas as linhas do `data.frame` gerado, o objeto `handsuit`, e recorrendo novamente à função **`unique()`**, obtém-se um objeto da classe `list`, onde cada um dos componentes é um objeto (`vector`) da classe `character`, correspondente a cada uma das linhas do objeto `handsuit` mas sem caracteres repetidos (`unisuit`).
Assim, se uma linha do objeto `handsuit` for constituída pelos caracteres `'C D C H C'`, então, a lista `unisuit` considera apenas os caracteres `'C D H'`. Posteriormente, obtém-se o comprimento de cada um dos seus componentes (`lensuit`). Se o comprimento for um, então, no objeto `handsuit`, as linhas correspondentes são constituídas por cinco elementos com os mesmos caracteres. É criado o objeto `flushtot`, da classe `data.frame`, que considera os casos em que as cartas são todas do mesmo naipe:

```
1 unisuit<-apply(handsuit,1,unique)
2 lensuit<-sapply(unisuit,length)
3 flushtot<-subset(hand,lensuit==1)
```

`output:`

```
> head(flushtot)
  card.1 card.2 card.3 card.4 card.5    probs
1    2 C    3 C    4 C    5 C    6 C 3.85e-07
2    2 C    3 C    4 C    5 C    7 C 3.85e-07
3    2 C    3 C    4 C    5 C    8 C 3.85e-07
4    2 C    3 C    4 C    5 C    9 C 3.85e-07
5    2 C    3 C    4 C    5 C   10 C 3.85e-07
6    2 C    3 C    4 C    5 C    J C 3.85e-07
```

De seguida, é necessário distinguir as três primeiras *mãos* mencionadas anteriormente: *Flush*, *Straight flush* e *Royal flush*.
No que diz respeito à *mão Royal flush*, apenas existem quatro casos favoráveis associados a este acontecimento, sendo relativamente fáceis de obter. Para tal, basta percorrer as linhas do objeto `flushtot` e retirar os quatro casos correspondentes.
De notar que, voltando ao objeto `handnum`, este associa aos elementos *J, Q, K* e *A* os valores numéricos 11, 12, 13 e 14, para que também seja possível obter as sequências ordenadas existentes no jogo. Neste caso, as sequências pretendidas têm como elemento

mínimo o valor numérico 10 e elemento máximo o valor numérico 14 (*Ás*). Assim, basta recorrer ao valor mínimo e máximo de um conjunto de cinco cartas (do mesmo naipe) para obter a *mão* pretendida:

```
flushsuit<-subset(handnum,lensuit==1)
numflush<-t(apply(flushsuit,2,as.numeric))
minf<-apply(numflush,2,min)
maxf<-apply(numflush,2,max)
Royalflush<-subset(flushtot,minf==10 & maxf==minf+4)
```

output:

```
> Royalflush
        card.1 card.2 card.3 card.4 card.5    probs
1512953   10 C    J C    Q C    K C    A C 3.85e-07
2429050   10 D    J D    Q D    K D    A D 3.85e-07
2590393   10 H    J H    Q H    K H    A H 3.85e-07
2598960   10 S    J S    Q S    K S    A S 3.85e-07
```

Em relação à *mão Straight flush*, o processo é semelhante, diferindo apenas no número de sequências ordenadas a considerar. Neste caso, são incluídas todas as sequências ordenadas de cinco cartas, do mesmo naipe, existentes no jogo, excepto as referentes á *mão royalflush*. Um método eficiente para excluir estes casos consiste em recorrer à função **merge()** e **cbind()** para associar as linhas do objeto `Royalflush` a um novo objeto (`seqf`), também da classe `data.frame`. Posteriormente, é gerado um objeto (`noroyal`), constituído por todas as sequências ordenadas do mesmo naipe, excepto as correspondentes a um *Royal flush*:

```
seqf<-hand[rownames(subset(flushsuit,minf==minf & maxf==minf+4))
   ,]
linesroyal<-merge(cbind(seqf,rownum=1:nrow(seqf)),Royalflush)$
   rownum
noroyal<-seqf[-linesroyal,]
```

Excluídas as linhas correspondentes a um *Royal flush*, o passo seguinte consiste em incluir a sequência ordenada baixa. Sabendo que, nesta vertente do *Poker*, o *Ás* pode assumir valor máximo ou mínimo, sendo possivel obter uma sequência ordenada do tipo *'A,2,3,4,5'* ou *'10, J, Q, K, A'*, facilmente se percebe que o número de casos existentes também é quatro, como no *Royal flush*, sendo necessário incluir esses casos. Para tal, é utilizada a função **`setequal()`**, onde são percorridas as linhas do objeto `flushsuit` e selecionadas aquelas que apresentam os elementos *2,3,4,5,14*.
Posteriormente, é criado o novo objeto `Straightflush`, da classe `data.frame`, com todos os casos favoráveis à *mão* indicada:

```
strlow<-which(apply(flushsuit,1,setequal,c(2,3,4,5,14)))
Straightflush<-rbind(noroyal,flushtot[strlow,])
```

`output:`

```
> head(Straightflush)
        card.1 card.2 card.3 card.4 card.5    probs
1          2 C    3 C    4 C    5 C    6 C 3.85e-07
249901     3 C    4 C    5 C    6 C    7 C 3.85e-07
480201     4 C    5 C    6 C    7 C    8 C 3.85e-07
692077     5 C    6 C    7 C    8 C    9 C 3.85e-07
886657     6 C    7 C    8 C    9 C   10 C 3.85e-07
1065022    7 C    8 C    9 C   10 C    J C 3.85e-07
```

Encontrados os casos favoráveis às *mãos Straight flush* e *Royal flush*, o último passo consiste em obter os casos favoráveis à *mão Flush*, isto é, obter cinco cartas do mesmo naipe que não sejam sequências ordenadas. Para tal, basta excluir, de forma análoga ao caso anterior, as linhas correspondentes às *mãos Straight flush* e *Royal flush*:

```
flushlow<-flushtot[-strlow,]
linessf<-merge(cbind(flushlow,rownum=1:nrow(flushlow)),seqf)$
   rownum
Flush<-hand[rownames(flushlow[-linessf,]),]
```

output :

```
> head(Flush)
 card.1 card.2 card.3 card.4 card.5     probs
2    2 C    3 C    4 C    5 C    7 C 3.85e-07
3    2 C    3 C    4 C    5 C    8 C 3.85e-07
4    2 C    3 C    4 C    5 C    9 C 3.85e-07
5    2 C    3 C    4 C    5 C   10 C 3.85e-07
6    2 C    3 C    4 C    5 C    J C 3.85e-07
7    2 C    3 C    4 C    5 C    Q C 3.85e-07
```

De todas as *mãos* existentes neste jogo, o método utilizado para obter os casos referentes à *mão Straight* exige um pouco mais de raciocínio. Ainda assim, o processo é semelhante aos casos anteriores, com a particularidade de se considerar todas as sequências ordenadas existentes no jogo, excepto os casos em que as cartas são do mesmo naipe (*Straight flush* e *Royal flush*).

Tendo em conta que é necessário incluir as sequências altas e baixas, neste caso, de naipes diferentes, um método eficiente para obter os casos referentes a esta *mão* é semelhante ao utilizado para obter um *Straight flush*.

Para tal, é necessário especificar o comprimento (`len`) dos componentes do objeto `uni`. Isto porque, considerando apenas o mínimo e o máximo dos elementos de cada linha do objeto `handnum`, serão também consideradas as linhas que admitem valores numéricos repetidos. Por exemplo, a linha composta pelos caracteres `'3 C,3 D,5 C,6 C,7 H'` tem como valor mínimo o caractere `'3 C'` e como valor máximo o caractere `'7 C'` e, no entanto, não se trata de uma sequência.

Uma forma de excluir estes casos consiste em definir `len=5`, ou seja, considerar as linhas do objeto `handnum` constituídas por cinco valores numéricos distintos e, posteriormente, aplicando o mínimo e o máximo a estes casos, obtêm-se as sequências ordenadas pretendidas. No caso das *mãos Straight flush* e *Royal flush*, ao considerar apenas as cartas do mesmo naipe, esta restrição já exclui as cartas com o mesmo valor, pois não existem duas cartas do mesmo naipe com o mesmo valor numérico. Deste modo, tem-se:

```
1 mini<-apply(handnum,1,min)
2 maxi<-apply(handnum,1,max)
```

```
straigh_high<-hand[rownames(subset(handnum,len==5 & mini==mini &
    maxi==mini+4)),]
straight_low<-hand[which(apply(handnum,1,setequal,c(2:5,14))),]
straighttot<-rbind(straight_high,straight_low)
```

Visto que se pretende considerar apenas as sequências ordenadas cujas cartas não são todas do mesmo naipe, é necessário excluir esses casos, ou seja, retirar as linhas correspondentes ás *mãos Straight flush* e *Royal flush*:

```
st<-merge(cbind(straighttot,rownum=1:nrow(straighttot)),flushtot
   )$rownum
Straight<-straighttot[-st,]
```

output:

```
> head(Straight)
   card.1 card.2 card.3 card.4 card.5    probs
14    2 C    3 C    4 C    5 C    6 D 3.85e-07
27    2 C    3 C    4 C    5 C    6 H 3.85e-07
40    2 C    3 C    4 C    5 C    6 S 3.85e-07
60    2 C    3 C    4 C    6 C    5 D 3.85e-07
73    2 C    3 C    4 C    6 C    5 H 3.85e-07
86    2 C    3 C    4 C    6 C    5 S 3.85e-07
```

Por fim, falta referir a última *mão*, isto é, os casos que não se encontram em nenhuma das *mãos* obtidas até ao momento. Esta *mão*, *High card*, é a menos valiosa de todas pois tem maior probabilidade de sair do que todas as outras. Os casos favoráveis associados a este acontecimento são facilmente encontrados, bastando, para tal, criar um novo objeto (`all_hands`), da classe `data.frame`, que reúna todas as *mãos* obtidas até ao momento e, comparando com o objeto de maior dimensão criado inicialmente, o objeto `hand`, sejam extraídas as linhas correspondentes aos casos que não se encontram no objeto `all_hands`:

```
allhands<-rbind(Royalflush,Straightflush,Four_kind,Fullhouse,
   Flush,Straight,Three_kind,Two_pairs,Pair)
alllines<-which((rownames(hand)%in%rownames(allhands)))
Highcard<-hand[-alllines,]
```

output :

```
> head(Highcard)
   card.1 card.2 card.3 card.4 card.5    probs
15    2 C    3 C    4 C    5 C    7 D 3.85e-07
16    2 C    3 C    4 C    5 C    8 D 3.85e-07
17    2 C    3 C    4 C    5 C    9 D 3.85e-07
18    2 C    3 C    4 C    5 C   10 D 3.85e-07
19    2 C    3 C    4 C    5 C    J D 3.85e-07
20    2 C    3 C    4 C    5 C    Q D 3.85e-07
```

Depois de definidos os acontecimentos referentes a cada uma das *mãos* consideradas, é criado o objeto `listhands`, da classe `list`, com dez componentes, da classe `data.frame`, referentes a cada *mão* existente. Cada componente tem ainda o nome da *mão* correspondente. Por exemplo, se se pretender obter a *mão Royal flush*, basta selecionar o componente da lista correspondente:

```
listhands<-list(Royalflush,Straightflush,Four_kind,Fullhouse,
   Flush,Straight,Three_kind,Two_pairs,Pair,Highcard)
names(listhands)<-c("Royalflush","Straightflush","Four_kind","
   Fullhouse","Flush","Straight","Three_kind","Two_pairs","Pair"
   ,"Highcard")
rl<-listhands$Royalflush
```

output :

```
> rl
       card.1 card.2 card.3 card.4 card.5    probs
```

```
1512953   10 C    J C    Q C    K C    A C 3.85e-07
2429050   10 D    J D    Q D    K D    A D 3.85e-07
2590393   10 H    J H    Q H    K H    A H 3.85e-07
2598960   10 S    J S    Q S    K S    A S 3.85e-07
```

4.3.2 Obtenção das probabilidades associadas a cada *mão*

O passo seguinte consiste em criar o objeto numérico `handtype`, referente aos nomes atribuídos a cada uma das *mãos* existentes. Para o efeito, são utilizadas as funções **`sapply()`** e **`rep()`** para que esses nomes se repitam tanto quanto o número de casos favoráveis a cada uma das *mãos*.
Posteriormente, para uma melhor visualização dos dados, o objeto `listhands` é convertido num objeto da classe `data.frame`, sendo utilizada a funcional **`do.call()`**. Esta função constrói e executa uma função a partir de outra e aplica-a num objeto da classe `list`. Neste caso os argumentos são a função **`rbind()`** e o objeto `listhands`, gerando o objeto `hands`.
É ainda utilizada a função **`gsub()`**, com o argumento `pattern='\\D'`, para eliminar os caracteres atribuídos aos nomes das linhas e, posteriormente, é imposta uma ordenação conveniente às linhas do objeto `hands` (para que tenha a mesma ordem que as linhas do objeto `hand`, criado inicialmente), através da função **`order()`**:

```
1 handtype<-rep(names(listhands),sapply(listhands,dim)[1,])
2 hands<-cbind(do.call(rbind,listhands),handtype)
3 nochar<-as.numeric(gsub("\\D","",rownames(hands)))
4 hands<-hands[order(nochar),]
5 rownames(hands)<-sort(nochar)
```

`output:`

```
> head(hands)
  card.1 card.2 card.3 card.4 card.5    probs      handtype
1    2 C    3 C    4 C    5 C    6 C 3.85e-07 Straightflush
2    2 C    3 C    4 C    5 C    7 C 3.85e-07         Flush
```

```
3    2 C    3 C    4 C    5 C    8 C 3.85e-07         Flush
4    2 C    3 C    4 C    5 C    9 C 3.85e-07         Flush
5    2 C    3 C    4 C    5 C   10 C 3.85e-07         Flush
6    2 C    3 C    4 C    5 C    J C 3.85e-07         Flush
```

O espaço de probabilidade do jogo de *Poker* está assim definido no objeto `hands`. É então possível recorrer às funções **`subset()`** e **`Prob()`** constantes do *package* `prob` para obter os subconjuntos referentes a cada uma das *mãos* e respectivas probabilidades associadas.
Em particular, para obter a probabilidade associada a qualquer uma das *mãos*, basta recorrer à função **`Prob()`** e definir o argumento `event` com o nome da *mão* pretendida, em vez de aplicar a função num subconjunto predefinido. Caso se pretenda obter todas as probabilidades, pode ser utilizada a função **`sapply()`** no objeto `listhands`.

Exemplo 4.3.1.

```
pair<-subset(hands,handtype=="Pair")
pairprob1<-Prob(pair)
pairprob2<-Prob(hands,event=handtype=="Pair")
probs<-as.matrix(sapply(listhands,Prob))
```

`output`:

```
> pairprob1
[1] 0.422569

> pairprob2
[1] 0.422569

> probs
                      [,1]
Royalflush    1.539077e-06
Straightflush 1.385169e-05
Four_kind     2.400960e-04
```

```
Fullhouse     1.440576e-03
Flush         1.965402e-03
Straight      3.924647e-03
Three_kind    2.112845e-02
Two_pairs     4.753902e-02
Pair          4.225690e-01
Highcard      5.011774e-01
```

Nota: Devido ao número elevado de casos possíveis deste jogo, que resulta num espaço de probabilidade consideravelmente extenso, com o intuito de economizar tempo na avaliação dos códigos criados, sãos disponibilizados *links* onde é possível aceder ao ambiente trabalho (*workspace*) em R, referente a todos os processos utilizados na obtenção deste espaço de probabilidade. Para tal, basta efetuar o *download* do ficheiro *'poker.Rdata'* disponível na partilha (Anexo A).

4.4 Simulação do jogo: `Poker()`

Definido o espaço de probabilidade do jogo em estudo, falta apenas testar o jogo, ou seja, jogar. Para que tal seja possível, nesta subsecção, é criada uma simulação do jogo.
A primeira ideia que surge é a de utilizar a função **`sim()`**, no entanto, não seria adequado, pois, no caso do jogo de *poker*, as cartas retiradas não são repostas no baralho e, como tal, todas as linhas têm de apresentar cartas diferentes entre si.

Exemplo 4.4.1.

Supondo que se pretende selecionar, aleatoriamente, cinco linhas do objeto `hands`, equivalente à experiência aleatória de selecionar cinco jogadores e verificar a *mão* correspondente, tem-se:

```
1 simpoker<-sim(hands,5)
```

`output:`

```
> simpoker
  card.1 card.2 card.3 card.4 card.5 handtype
```

```
1    4 C    K D    2 H    5 S    9 S Highcard
2    8 D    5 H    7 H   10 S    K S Highcard
3    K C    5 D   10 D    4 S    A S Highcard
4    J C    7 D   10 H    A H    9 S Highcard
5    3 C    6 D    9 D    K D    9 H     Pair
```

Numa primeira estância, aparentemente, a simulação é bem conseguida. No entanto, não é este tipo de simulação que se pretende. Repare-se que, por exemplo, a carta *'9 S'* aparece na linha um e quatro. As cartas obtidas pelo primeiro jogador não podem voltar a ser consideradas no baralho. Ou seja, o jogador seguinte recebe cinco cartas sabendo que já saíram cinco cartas ao jogador anterior e, como tal, não podem haver cartas repetidas. Trata-se, portanto, de probabilidade condicionada e, consequentemente, este tipo de processo não pode ser aplicado.
Assim, o método encontrado para simular este jogo consiste em retirar aleatoriamente um número específico de cartas do baralho criado inicialmente (`deck`) de acordo com o número (n) de jogadores considerado. Dado que uma *mão* é composta por cinco cartas e que são necessários entre dois a dez jogadores (isto porque o baralho tem cinquenta e duas cartas, e dez jogadores implica retirar cinquenta cartas do baralho) para iniciar o jogo, considera-se um máximo de dez jogadores (*Player 1 a Player 10*) e, consequentemente, são retiradas $5 \times n$ cartas do baralho.
Recorrendo à função **`sample()`** são então retiradas, aleatoriamente, $5 \times n$ cartas do baralho e distribuídas pelos n jogadores.

Exemplo 4.4.2.

A título de exemplo, consideram-se oito jogadores. Tem-se:

```
1 n<-8
2 sampling<-str_trim(stri_sub(sample(deck,5*n),-4))
3 plyrs<-matrix(sampling,ncol=5)
```

`output:`

```
> plyrs
     [,1]  [,2]   [,3]  [,4]  [,5]
[1,] "4 D" "8 D"  "8 H" "9 S" "Q S"
[2,] "9 C" "9 D"  "Q H" "4 S" "K S"
[3,] "5 D" "7 H"  "5 S" "6 S" "J S"
[4,] "5 C" "10 C" "K C" "2 H" "J H"
[5,] "Q C" "3 D"  "K H" "A H" "7 S"
[6,] "3 C" "J D"  "Q D" "9 H" "10 H"
[7,] "8 C" "A D"  "3 H" "2 S" "A S"
[8,] "6 C" "5 H"  "6 H" "8 S" "10 S"
```

Do Exemplo 4.4.2, facilmente se percebe que cada linha do objeto `plyrs`, da classe `matrix`, corresponde à *mão* obtida por cada jogador.
Após a distribuição das cartas, é utilizada a função **`setequal()`**, para estabelecer uma correspondência entre as *mãos* de cada um dos jogadores com as linhas do objeto `hands`, como se pode verificar no exemplo seguinte:

Exemplo 4.4.3.

```
1 matchs<-hands[which(apply(hands[,1:5],1,setequal,plyrs[1,])),]
```

`output:`

```
> matchs
       card.1 card.2 card.3 card.4 card.5   probs handtype
753050    5 C   10 C    K C    2 H    J H 3.85-07 Highcard
```

O Exemplo 4.4.3 representa apenas a *mão* de um dos jogadores. Para obter as restantes *mãos*, é aplicado um ciclo **`for`** para obter todas as correspondências:

Exemplo 4.4.4.

```
Plyrs<-list()
for (i in 1:dim(plyrs)[1]){
    Plyrs[i]=list(hands[which(apply(hands[,1:5],1,setequal,plyrs
       [i,])),])
}
Plyrs<-cbind(Players=paste("Players",1:n),do.call(rbind,Plyrs))
```

output:

```
> Plyrs
         Players card.1 card.2 card.3 card.4 card.5    probs handtype
2185027 Players 1    4 D    8 D    8 H    9 S    Q S 3.85e-07     Pair
1480986 Players 2    9 C    9 D    Q H    4 S    K S 3.85e-07     Pair
2269368 Players 3    5 D    7 H    5 S    6 S    J S 3.85e-07     Pair
753050  Players 4    5 C   10 C    K C    2 H    J H 3.85e-07 Highcard
1783068 Players 5    Q C    3 D    K H    A H    7 S 3.85e-07 Highcard
453003  Players 6    3 C    J D    Q D    9 H   10 H 3.85e-07 Highcard
1360161 Players 7    8 C    A D    3 H    2 S    A S 3.85e-07     Pair
1056357 Players 8    6 C    5 H    6 H    8 S   10 S 3.85e-07     Pair
```

Por último, é necessário definir o vencedor após a comparação entre as *mãos* de cada jogador. Quando se tratam de cartas associadas a *mãos* distintas, o processo é simples, bastando recorrer ao valor de probabilidade de cada *mão* e escolher a que tiver menor probabilidade. No entanto, quando vários jogadores apresentam cartas associadas ao mesmo tipo de *mão* (da mesma classe) o processo torna-se um pouco mais complicado.

Exemplo 4.4.5.

No Exemplo 4.4.4, os jogadores obtiveram cartas referentes às *mãos Pair* e *High card*. Facilmente se percebe que os jogadores que obtiveram *High card* perdem relativamente aos que obtiveram *Pair*. Assim, aproveitando o ciclo criado anteriormente, tem-se:

```
Plyrs<-list()
who_wins<-c()
for (i in 1:dim(plyrs)[1]){
```

```
    Plyrs[i]=list(hands[which(apply(hands[,1:5],1,setequal,plyrs
        [i,])),])
    who_wins[i]=c(Prob(hands,handtype==Plyrs[[i]]$handtype))
}
Plyrs<-cbind(Players=paste("Players",1:n),do.call(rbind,Plyrs))
winners<-Plyrs[which(who_wins==min(who_wins)),]
```

output:

```
        Players card.1 card.2 card.3 card.4 card.5      probs handtype
2185027 Players 1    4 D    8 D    8 H    9 S    Q S 3.85e-07     Pair
1480986 Players 2    9 C    9 D    Q H    4 S    K S 3.85e-07     Pair
2269368 Players 3    5 D    7 H    5 S    6 S    J S 3.85e-07     Pair
1360161 Players 7    8 C    A D    3 H    2 S    A S 3.85e-07     Pair
1056357 Players 8    6 C    5 H    6 H    8 S   10 S 3.85e-07     Pair
```

O passo seguinte consiste, portanto, em decidir o vencedor entre os cinco jogadores que obtiveram um *Pair*. No Exemplo 4.4.5, o vencedor é o *player 7* pois apresenta um *Pair* mais alto (par de ases) que os restantes.
Assim, para abranger todo o tipo de decisões possíveis relativamente ao vencedor é criada a função **`poker()`** (disponível no Anexo B). Esta função tem como único argumento:

- `n` - número de jogadores iniciais.

O *output* desta função é um objeto, da classe `list` e é constituído pelos seguintes componentes:

- um objeto de caracteres ou *string* com o resultado do jogo;
- um objeto, da classe `data.frame`, cujas linhas correspondem às cinco cartas que saíram a cada jogador, com a informação adicional do tipo de *mão*, associado à cartas obtidas.

De notar que o resultado do *output* não será sempre o mesmo, tendo em conta que se trata de um simulação. De seguida são testados alguns resultados possíveis do *output* desta função:

output:

```
> poker(n=2)
  |======================================================================| 100%
 Winner: Player 1
$Hands
          Players card.1 card.2 card.3 card.4 card.5 hands
1960094 Players 1    A C    4 D   10 D   10 H    6 S  Pair
227326  Players 2    2 C    Q D    3 H    2 S    5 S  Pair

> poker(n=8)
  |======================================================================| 100%
 Winner: Player 7
$Hands
          Players card.1 card.2 card.3 card.4 card.5     hands
1122582 Players 1    7 C    Q C   10 D    3 H    9 H  Highcard
61347   Players 2    2 C    6 C    K C    2 H    6 S Two_pairs
690989  Players 3    4 C    K H    5 S    Q S    K S      Pair
2168263 Players 4    4 D    5 D    6 H    8 S    A S  Highcard
1554396 Players 5   10 C    2 D    3 D    Q D    J S  Highcard
325277  Players 6    3 C    8 C    7 D    9 D    7 S      Pair
769516  Players 7    5 C    J C    8 D    5 H    8 H Two_pairs
2300291 Players 8    6 D    K D    A D    4 H    4 S      Pair

> poker(n=11)
The number of players shouldn´t be more than 10 or less than 2

> poker(n=1)
The number of players shouldn´t be more than 10 or less than 2
```

- No primeiro exemplo , foi feita uma simulação para dois jogadores. Ambos obtiveram *Pair*, no entanto, o *Player 1* tem o *Pair* mais alto.

- No segundo exemplo, a simulação foi feita para oito jogadores, obtendo-se três *High card*, três *Pair* e dois *Two pairs*. O jogador com a *mão* mais valiosa, neste caso, é o *Player 7* com o *Two pairs* mais alto.

- Nos restantes exemplos foi introduzido um número de jogadores não aplicável pois foi definido que o mínimo e máximo de jogadores seria dois e dez, respectivamente.

Quando dois ou mais jogadores apresentam *mãos* da mesma classe, os critérios de desempate mais complicados de aplicar são:

- *Straight* de *Ás baixo* com outro *Straight*;

- *Straight flush* de *Ás baixo* com outro *Straight flush*;

Isto porque, como foi explicado anteriormente, o *Ás* pode assumir valor máximo ou mínimo. Por este facto, é necessário realçar que, para estes casos, o jogador que tenha um *Ás* não ganha necessariamente o jogo. Se a sequência obtida, *Straight* ou *Straight flush*, for de *Ás baixo* (o *Ás* assume valor 1), esta é considerada a sequência mais baixa de todas e, consequentemente, a menos valiosa. Contrariamente, em todas as outras *mãos* o *Ás* assume sempre valor máximo, 14, sendo este o critério de desempate nestes casos.

No entanto, a função **`poker()`** consegue aplicar todos os critérios de desempate necessários. Por exemplo, entre dois *Straights* (um deles de *Ás baixo*), tem-se:

```
> poker(n=2)
  |======================================================================| 100%
 Winner: Player 2
$Hands
          Players card.1 card.2 card.3 card.4 card.5 handtype
165350  Players 1    2 C    A C    3 H    4 H    5 H Straight
1818068 Players 2    Q C    9 D    8 S   10 S    J S Straight
```

Esta função contempla ainda uma barra de progresso, onde é possível acompanhar a percentagem de execução do código, bem como verificar o tempo decorrido na avaliação do mesmo:

```
> system.time(print(poker(2)))
  |======================================================================| 100%
 Winner: Player 1
$Hands
          Player card.1 card.2 card.3 card.4 card.5 handtype
1006615 Player 1    6 C    5 D    6 D    3 S    7 S     Pair
1498141 Player 2    9 C    2 H    3 H   10 H    K S Highcard

   user  system elapsed
  37.67    0.58   39.00
```

Capítulo 5

Conclusão

Neste trabalho foram exploradas algumas potencialidades da linguagem `R`, direcionada à teoria de probabilidades. Numa primeira instância foram explicados conceitos importantes para uma melhor compreensão do tema. Em particular, o *package* `prob`, onde foram abordadas algumas funções fundamentais para a elaboração deste trabalho, bem como a possibilidade de criar novas funções para obter novos espaços de probabilidade, nomeadamente, de um jogo de *Poker*.

De seguida, foram introduzidos alguns conceitos relativos ao jogo, tais como: *mãos* existentes, regras, probabilidades, etc.

Posteriormente foi criado o espaço de probabilidade do jogo, sendo possível extrair os subconjuntos referentes às *mãos* existentes, bem como as probabilidades associadas à mesmas. Todo o processo foi conseguido utilizando funções já existentes em `R`, evitando assim métodos, eventualmente, mais complexos e demorados (ciclos **`for`**, por exemplo). Devido ao número consideravelmente grande de casos possíveis do jogo, resultando num espaço de probabilidade muito grande (o objeto `hands`, da classe `data.frame`, é constituído por mais de 2.5 milhões de linhas), a avaliação dos processos utilizados para obter este espaço de probabilidade não é de todo rápida, ainda assim, incomparável ao tempo de avaliação se fossem utilizados ciclos **`for`**. Daí os *links* disponibilizados no Anexo A para poder aceder ao espaço de probabilidade sem precisar de executar a avaliação dos processos.

No final, foi criada a função **`poker()`** que gera uma simulação do jogo, obedecendo a todas as regras inerentes ao mesmo. Em termos do jogo em si, esta função aplica uma

ronda inicial de *n* jogadores e decide o vencedor com base nas *mãos* obtidas, não sendo considerada a possibilidade de fazer apostas ou trocar cartas, devido à subjectividade e complexidade destes processos. A decisão de um jogador apostar ou trocar cartas não é, de todo, um processo controlável. Não seria justo decidir este tipo de situações com base na aleatoriedade. Como tal, a função criada restringe-se ao tipo de *mão* que cada jogador apresenta inicialmente, decidindo o vencedor com base no valor de cada *mão*.

Possivelmente, existem programas mais eficientes que os apresentados neste trabalho, mas o desafio era utilizar a linguagem `R`, evitando ciclos, e usando funções de `R`, como por exemplo as da família **`apply()`**, para melhorar o desempenho dos programas propostos. De notar que na função implementada, **`Poker()`**, foi usado apenas um ciclo **`for`**.

Assim, a sugestão passa por melhorar o desempenho dos códigos criados sem, naturalmente, descaracterizar o trabalho realizado.

Bibliografia

Chambers, J.M. (2008) Software for Data Analysis: Programming with R, Springer, New York.

DataCamp, Functions in R (2016), https://www.datacamp.com/community/tutorials/functions-in-r-a-tutorial#gs.DiL6NuY

Peng, Roger D. (2015) R programming for Data science, https://leanpub.com/rprogramming

Sanchez, G. (2013) Handling and Processing Strings in R, http://www.gastonsanchez.com/HandlingandProcessingStringsinR.pdf, Trowchez Editions. Berkeley, 2013.

Wickham, H. (2010) stringr: modern, consistent string processing, https://journal.r-project.org/archive/2010-2/RJournal_2010-2_Wickham.pdf

Kerns, J. (2013) Elementary Probability and the prob Package, Department of Mathematics and Statistics, Youngstown State University, http://gkerns.people.ysu.edu/

Coffin, George Sturgis, Secrets of Winning Poker, Youngstown State University, Wilshire Book Company, 1949

Conover, W.J.(1999), Practical nonparametric statistics, Third Edition, New York: John Wiley & Sons.

CRAN, R Development Core Team. (2008), R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria, http://www.R-project.org

Jornal Público, 25 de Setembro (2013), http://p3.publico.pt/actualidade/desporto/9405/primeiro-campeao-mundial-de-poker-portugues

Wikipedia, History of poker(2016), https://en.wikipedia.org/wiki/History_of_poker

jogapoker, As regras do Poker 5 Card Draw(2009), http://www.jogapoker-online.com/regras-poker/5-card-draw/

Ramsey, T. (2005), 5-CARD POKER HANDS(2005), UHM Department of Mathematics http://www.math.hawaii.edu/~ramsey/Probability/PokerHands.html

ANEXOS

Anexo A

Neste anexo encontram-se os *links* disponíveis para obter o ambiente de trabalho do jogo em R, bem como os *packages* necessários para a realização do mesmo:

- https://1drv.ms/u/s!AqcTYaaLZ814nn_uT4idlI3aU1cw
- https://www.dropbox.com/s/r0kmwzcc1e2o41c/poker.RData?dl=0
- *package* stringi
- *package* stringr
- *package* prob

Após o *download* do ficheiro, chamado *poker.Rdata*, é necessário definir a diretoria de acordo com a localização do ficheiro no computador. Para o efeito, é utilizada a função **setwd()** para definir a diretoria. Posteriormente, a função **load()** permite abrir o ambiente de trabalho pretendido. A título de exemplo, se o ficheiro estiver no *desktop* do computador, a diretoria deverá ser algo do género:

```
1 setwd("C:\\Users\\Desktop")
2 load(file='poker.Rdata")
```

Anexo B

Neste anexo encontra-se o código da função **poker()**:

```
poker<-function(n){
if(n<1 & n>11){
    sampling<-str_trim(stri_sub(sample(deck,5*n),-4))
    plyrs<-matrix(sampling,ncol=5)
    Plyrs<-list()
    who_wins<-c()
    Time<-txtProgressBar(min=0,max=dim(plyrs)[1],style=3)
    for (i in 1:dim(plyrs)[1]){
        Plyrs[i]=list(hands[which(apply(hands[,1:5],1,setequal,
           plyrs[i,])),])
        who_wins[i]=c(Prob(hands,handtype==Plyrs[[i]]$handtype))
        setTxtProgressBar(Time,i)
    }
    Plyrs<-cbind(Player=paste("Players",1:n),do.call(rbind,Plyrs
       ))
    winners<-Plyrs[which(who_wins==min(who_wins)),]
    if (dim(winners)[1]>1){
        winners2<-handnum[rownames(winners),]
        unisim<-apply(winners2,1,unique)
        max1<-apply(winners2,1,max)
        vals<-apply(winners2,1,sum)-apply(unisim,2,sum)
        if (sum(vals)==0){
            if(unique(winners$handtype=='Straight'|winners$
               handtype=='Straightflush')){
                winners3<-apply(winners2,1,sort)[-5,]
                max2<-apply(winners3,2,max)
                fiwin<-winners[which(max2==max(max2)),]
            }
            else
```

```
                fiwin<-winners[which(max1==max(max1)),]
        }
        else
            fiwin<-winners[which(vals==max(vals)),]
            if (dim(fiwin)[1]!=1){
                cat("\n","Tie between:","\n", as.character(fiwin
                   $Player),"\n","Winner : The player with the
                   highest card","\n")
            }
            else
                cat("\n","Winner:", as.character(fiwin$Player),"
                   \n")
    }
    else
        cat("\n","Winner:", as.character(winners$Player),"\n")
    list(Hands=Plyrs[,-7])
}
else
    cat("The number of players shouldn't be more than 8 or less
       than 2","\n")
}
```